Large Language Models

Paul He

September 2, 2024

Notes are based on LLM course notes and lectures at ETH Zürich (some parts may be identical in terms of wording). These are simply my notes used for studying the course. I do not guarantee the correctness of this set of notes, please feel free to point out if there are typos or mistakes.

Contents

Part I

Theory

1 Probabilistic Foundations

1.1 Language Modeling

Language Model (Informal)

Given an alphabet Σ (*finite, non-empty*) and a distinguished end-of-sequence symbol $\textsc{eos} \notin \Sigma$, a language model is a collection of conditional probability distributions $p(y|\boldsymbol{y})$ for $y \in \Sigma \cup \{\textsc{eos}\}$, and $\boldsymbol{y} \in \Sigma^*$. $p(y|\boldsymbol{y})$ therefore represents the probability of y being the next token given the history $\boldsymbol{y}$.

Most papers have the following autoregressive factorization:

$$p(\boldsymbol{y}) = p(y_1 \dots y_T) = p(\textsc{eos}|\boldsymbol{y}) \prod_{t=1}^{T} p(y_t|\boldsymbol{y}_{<t}) \tag{1}$$

where $\boldsymbol{y} \in \Sigma^*, y_t \in \overline{\Sigma}$ and $\boldsymbol{y}_{<t} \in \Sigma^*$

It is left implicit, whether or not p is indeed a valid probability distribution, and if it is, over what space. The natural assumption of the informal definition of Language Model's given above, is that p is a distribution over Σ^*. However, it is generally not true that all such collections of conditionals will yield a valid probability distribution over Σ^*, as some may "leak" probability mass to infinite sequences. We additionally have to be very careful when dealing with uncountably infinite spaces lest we run into a classic paradox, which we will demonstrate below.

Infinite Toin-Coss Example

Consider the infinite, independent fair coin toss model, where we aim to place a distribution over $\{\texttt{H}, \texttt{T}\}^\infty$. Such distribution corresponds to a "language model", defined as for all $\boldsymbol{y}_{<t}, p(\texttt{H}|\boldsymbol{y}_{<t}) = p(\texttt{T}|\boldsymbol{y}_{<t}) = \frac{1}{2}$ and $p(\textsc{eos}|\boldsymbol{y}_{<t}) = 0$. However, with each individual sequence over $\{\texttt{H}, \texttt{T}\}$ should also be assigned probability $(\frac{1}{2})^\infty = 0$. Without a formal foundation, we arrive at the following

paradox:

$$1 = p(\{\texttt{H, T}\}^\infty) = p(\bigcup_{\omega \in \{\texttt{H, T}\}^\infty} \{\omega\})$$
$$= \sum_{\omega \in \{\texttt{H, T}\}^\infty} p(\{\omega\}) = \sum_{\omega \in \{\texttt{H, T}\}^\infty} 0 \stackrel{?}{=} 0$$

There is another example in the course notes, which I will omit here.

1.2 *A Measure-theoretic Foundation*

Let Ω be the **outcome space** and $\mathcal{F}$ be the set of **measurable subsets**

Probability Measure

Let $\mathbb{P} : \mathcal{F} \to [0,1]$ be the **probability measure**, which is a function that assigns probability to each measurable subset, where:

1. $\mathbb{P}(\Omega) = 1$
2. If $\varepsilon_1, \varepsilon_2$ is a countable sequence of disjoint sets in $\mathcal{F}$, then $\mathbb{P}(\bigcup_n \varepsilon_n) = \sum_n \mathbb{P}(\varepsilon_n)$

Then, $(\Omega, \mathcal{F}, \mathbb{P})$ is the **probability space**

Probability pre-measure

Let $\mathcal{A}$ be an Algebra over some set Ω, a **probability pre-measure** over $(\Omega, \mathcal{A})$ is a function $\mathbb{P}_0 : \mathcal{A} \to [0,1]$ such that

1. $\mathbb{P}_0(\Omega) = 1$
2. If $\varepsilon_1, \varepsilon_2, \ldots$ is a countable sequence of disjoint sets in $\mathcal{A}$ *whose countable union is also in* $\mathcal{A}$, then $\mathbb{P}_0(\bigcup_{n=1}^\infty \varepsilon_n) = \sum_{n=1}^\infty \mathbb{P}_0(\varepsilon_n)$

σ-algebra,

Let $\mathcal{P}(\Omega)$ be the power set of Ω. Then, $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is a σ-algebra over Ω iff:

1. $\Omega \in \mathcal{F}$
2. if $\varepsilon \in \mathcal{F}$, then $\varepsilon^\complement \in \mathcal{F}$
3. if $\varepsilon_1, \varepsilon_2 \ldots$ is a finite or infinite sequence in $\mathcal{F}$, then $\bigcup_n \varepsilon_n \in \mathcal{F}$

If $\mathcal{F}$ is a σ-algebra over Ω, we call the tuple $\Omega, \mathcal{F}$ a **measurable space.**

Let Ω be any set. Importantly, there is no one way to construct a σ-algebra over Ω:

1. The family consisting only the empty set $\varnothing$ and the set $\{\varnothing, \Omega\}$ is called the **minimal** or **trivial**.

2. The full power set $\mathcal{F} \stackrel{\text{def}}{=} \mathcal{P}(\Omega)$ is called the **discrete σ-algebra**.

3. Given $\mathcal{A} \subseteq \Omega$, the family $\mathcal{F} \stackrel{\text{def}}{=} \{\varnothing, \mathcal{A}, \Omega \backslash \mathcal{A}, \Omega\}$ is the σ-algebra induced by $\mathcal{A}$.

We can confirm these are indeed σ-algebra by checking them against the axioms.

Algebra

Let $\mathcal{P}(\Omega)$ be the power set of Ω. Then, $\mathcal{F} \subseteq \mathcal{P}(\Omega)$ is an algebra over Ω iff:
1. $\Omega \in \mathcal{F}$
2. if $\varepsilon \in \mathcal{F}$, then $\varepsilon^{\mathsf{c}} \in \mathcal{F}$
3. if $\varepsilon_1, \varepsilon_2 \in \mathcal{A}$, then $\varepsilon_1 \cup \varepsilon_2 \in \mathcal{A}$

The difference to σ-algebra, is that the definition is weakened from countable to finite.

Random

A mapping $x : \Omega \to \mathcal{S}$ between two measurable spaces $(\Omega, \mathcal{F})$ and $\mathcal{S}, \mathcal{T}$ is an $(\mathcal{S}, \mathcal{T})$-valued **random variable**, or a measurable mapping if for all $\mathcal{B} \in \mathcal{T}$,

$$x^{-1}(\mathcal{B}) \stackrel{\text{def}}{=} \{\omega \in \Omega : x(\omega) \in \mathcal{B}\} \in \mathcal{F} \tag{2}$$

1.3 *Defining a Language Model*

Language Model

Let Σ be an alphabet. A **language model** is a discrete distribution p_{LM} over Σ^*

As an example, we can construct a very simple language model. Let $\Sigma \stackrel{\text{def}}{=} \{a\}$, for $n \in \mathbb{N}_{\geq 0}$, define

$$p_{\text{LM}} \stackrel{\text{def}}{=} 2^{-(n+1)}$$

where, $a^0 = \varepsilon$ and $a^n = \underbrace{a \dots a}_{\text{n times}}$. To verify p_{LM} is a language model, we just check if the probabilities of finite sequences sum to 1:

$$\sum_{\boldsymbol{y} \in \Sigma^*} p_{\text{LM}}(\boldsymbol{y}) = \sum_{n=0}^{\infty} p_{\text{LM}}(a^n) = \sum_{n=0}^{\infty} 2^{-(n+1)} = \frac{1}{2} \sum_{n=0}^{\infty} \frac{1}{2^n} = \frac{1}{2} \frac{1}{1 - \frac{1}{2}} = 1$$

For this example, the **language** of p_{LM} is defined as

$$L(p_{\text{LM}}) \stackrel{\text{def}}{=} \left\{ \left(a^n, 2^{-(n+1)} \right) | n \in \mathbb{N}_{\geq 0} \right\} \tag{3}$$

With this, we can define the **weighted language**.

Weighted Language

Let p_{LM} be a language model, the **weighted language** of p_{LM} is defined as

$$L(p_{\text{LM}}) \stackrel{\text{def}}{=} \{(\boldsymbol{y}, p_{\text{LM}}(\boldsymbol{y})) | \boldsymbol{y} \in \Sigma^*\} \tag{4}$$

So, a language model is *simply a distribution that weights strings (natural utterances) by their probabilities to occur in a particular language.*

1.4 Globally Normalized Language Models

The next two chapters, discusses in depth the computational models which we can use to try to tractably represent distributions over strings and ways of approximating (learning) the ground-truth distribution based on finite datasets using such models.

An **energy function** is a function $\hat{p} : \Sigma^* \to \mathbb{R}$, we now can define a **globally normalized language model** in terms of an energy function over Σ^*

Globally Normalized Model

Let $\hat{p}_{\text{GN}}(\boldsymbol{y}) : \Sigma^* \to \mathbb{R}$, a **globally normalized model** (GNM) is defined as

$$p_{\text{LM}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y})\}}{\sum_{\boldsymbol{y}' \in \Sigma^*} \exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y}')\}} \stackrel{\text{def}}{=} \frac{1}{Z_G} \exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y})\} \tag{5}$$

- One only needs to define an (unnormalized) energy function $\hat{p}_{\text{GN}}$, which scores an entire sequence at once
- They define a probability distribution over strings $\boldsymbol{y} \in \Sigma^*$ directly
- However, Z_G can be expensive to compute!

Since Σ^* is infinite, Z_G might diverge to infinity! In this case, GNM is not defined. We say an energy function is **normalizable** if the quantity Z_G is finite. With this, we turn to a theorem

Normalizable energy functions induce language models

Any normalizable energy function p_{GN} induces a language model, i.e., a distribution over Σ^*.

Proof. Given an energy function $\hat{p}_{\text{GN}}$, we have $\exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y})\} \geq 0$ and

$$\sum_{\boldsymbol{y}\in\Sigma^*} p_{\text{GN}}(\boldsymbol{y}) = \sum_{\boldsymbol{y}\in\Sigma^*} \frac{\exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y})\}}{\sum_{\boldsymbol{y}'\in\Sigma^*} \exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y}')\}} \tag{6}$$

$$= \frac{1}{\sum_{\boldsymbol{y}'\in\Sigma^*} \exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y}')\}} \sum_{\boldsymbol{y}\in\Sigma^*} \exp\{-\hat{p}_{\text{GN}}(\boldsymbol{y})\} \tag{7}$$

$$= 1 \tag{8}$$

Which means, p_{GN} is a valid probability distribution over Σ^* □

1.5 *Locally Normalized Language Models*

Decomposes the problem into the problem of modeling a series of conditional distributions over the next possible symbol in the string given the context so far.

First, let's introduce the concept of prefix probabilities, which *denotes the sum of the probabilities of all strings beginning with a certain prefix*.

Prefix Probability

Let p_{LM} be a language model. We define a p_{LM}'s **prefix probability** π as

$$\pi(\boldsymbol{y}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{y}'\in\Sigma^*} p_{\text{LM}}(\boldsymbol{y}\boldsymbol{y}') \tag{9}$$

Any language model can be locally normalized

Let p_{LM} be a language model. Then, there exists a locally normalized language model p_{LN} such that, for all $\boldsymbol{y} \in \Sigma^*$ with $|\boldsymbol{y}| = T$,

$$p_{\text{LM}}(\boldsymbol{y}) = p_{\text{LN}}(\boldsymbol{y}) = p_{\text{SM}}(\text{EOS}|\boldsymbol{y}) \prod_{t=1}^{T} p_{\text{SM}}(y_t|\boldsymbol{y}_{<t}) \tag{10}$$

Proof. Let $y \in \Sigma, \boldsymbol{y} \in \Sigma^*$, assume $\pi(\boldsymbol{y}) > 0$ and define

$$p_{\text{LM}}(x|\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\pi(\boldsymbol{y}x)}{\pi(\boldsymbol{y})} \tag{11}$$

$$p_{\text{LM}}(\text{EOS}|\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{p(\boldsymbol{y})}{\pi(\boldsymbol{y})} \tag{12}$$

The idea is **telescoping product**

$$p_{\text{LM}}(\boldsymbol{y}) = p_{\text{LM}}(\text{EOS}|\boldsymbol{y}) \prod_{t=1}^{T} p_{\text{LM}}(y_t|\boldsymbol{y}_{<t}) \tag{13}$$

$$= \frac{p(\boldsymbol{y}_{<t+1})}{\pi(\boldsymbol{y}_{<t+1})} \prod_{t=1}^{T} \frac{\pi(\boldsymbol{y}_{<t+1})}{\pi(\boldsymbol{y}_{<t})} \tag{14}$$

$$= \frac{p(\boldsymbol{y}_{<t+1})}{\cancel{\pi(\boldsymbol{y}_{<t+1})}} \prod_{t=1}^{T} \frac{\cancel{\pi(\boldsymbol{y}_{<t+1})}}{\pi(\boldsymbol{y}_{<t})} \tag{15}$$

$$= p(\boldsymbol{y}_{<t+1})\pi(\varepsilon) \tag{16}$$

$$= p(\boldsymbol{y}) \tag{17}$$

□

1.6 Tight Language Models

Tightness

A locally normalized language model p_{LN} derived from a sequence model p_{SM} is called **tight** if it defines a valid probability distribution over Σ^*

$$\sum_{y \in \Sigma^*} p_{\text{LN}}(\boldsymbol{y}) = \sum_{y \in \Sigma^*} \left[p_{\text{SM}}(\text{EOS}|\boldsymbol{y}) \prod_{t=1}^{T} p_{\text{SM}}(y_t|\boldsymbol{y}_{<t}) \right] = 1 \tag{18}$$

Note that:

1. Individual conditional distributions $p_{\text{SM}}(y|\boldsymbol{y})$ in a non-tight LNM are still valid conditional distributions.
2. It is the ***distribution over all possible strings*** that they induce, that might not sum to 1 (invalid).
3. Given a sequence model p_{SM}, p_{LN} is a language model if the LNM's conditional probabilities match the conditional probabilities of a known language model p_{LM}, as any normalizable energy function induces a language model.

1.7 Defining the probability measure of an LNM

In this section, we want to define language models rigorously. If we want to answer "What is the probability that a language model generates a finite string?", we need infinite strings.

Re-definition of a Language Model

A **language** model is a probability space over Σ^*. Equivalently, a language model is a sequence model such that $\mathbb{P}(\Sigma^\infty) = 0$

Sequence Model

A **sequence model** is a probability space over the set $\Sigma^* \cup \Sigma^\infty$

The goal for this section is to rigorously construct a probability sequence model $\mathbb{P}(\Sigma^\infty) = 0$ to encode the probabilities assigned by a LNM. [1]

[1] Note to self, for Final Exam revision, ignore this section, but very relevant for Assignment 1.

Step 1: Defining an Algebra over $\overline{\Sigma}^\infty$ as the set of all infinite strings over $\Sigma \cup \{\text{EOS}\}$

Step 2: Define algebra over $\overline{\Sigma}^\infty$ using cylinder sets

Cylinder Set

Given any set $\mathcal{H} \subseteq \overline{\Sigma}^k$, define its **cylinder set** (of rank k) to be

$$\overline{C}(\mathcal{H}) \stackrel{\text{def}}{=} \left\{ \boldsymbol{y}\boldsymbol{\omega} | \boldsymbol{y} \in \mathcal{H}, \boldsymbol{\omega} \in \overline{\Sigma}^\infty \right\} \tag{19}$$

A cylinder set of rank k is basically *the set of infinite strings that share their k-prefix with some string* $\overline{\boldsymbol{y}} \in \mathcal{H} \subseteq \overline{\Sigma}^k$

We also denote the collection of all rank-k cylinder sets by

$$\overline{\mathcal{C}}_k \stackrel{\text{def}}{=} \left\{ \overline{C}(\mathcal{H}) | \mathcal{H} \in \mathcal{P}(\overline{\Sigma}^k) \right\} \tag{20}$$

and define the following to be the collection of all cylinder sets over Ω, where $\Omega = \overline{\Sigma}^\infty$

$$\overline{\mathcal{C}} \stackrel{\text{def}}{=} \bigcup_{k=1}^{\infty} \overline{\mathcal{C}}_k \tag{21}$$

We arrive at the following conclusion.

$\overline{\mathcal{C}} \subseteq \mathcal{P}(\Omega)$ is an algebra over $\Omega = \overline{\Sigma}^\infty$

Proof. We just have to check the lemma against the axioms according to the definition of an Algebra. Let $\Omega = \overline{\Sigma}^\infty$ and $\mathcal{F} = \overline{\mathcal{C}}$

1. For any k, we have $\Omega = \overline{\Sigma}^\infty = \overline{C}(\overline{\Sigma}^k) \in \overline{\mathcal{C}} = \mathcal{F}$
2. Given a cylinder set $\overline{C}(\mathcal{H})$ of rank k, we have $\overline{C}(\mathcal{H})^\complement = \overline{C}(\overline{\Sigma}^k \backslash \mathcal{H})$. Hence $\overline{\mathcal{C}}$ is closed under complements.

3. The union[2] of two cylinder sets of ranks $k_1 \leq k_2$ is another cylinder set of rank k_2

[2] double check, typo in course notes

□

Step 3: Defining a Pre-measure over $\overline{\mathcal{C}}$

Given an LNM p_{LN} and any set $\overline{C}(\mathcal{H}) \in \overline{C}$, define the pre-measure $\mathbb{P}_0$ for the cylinder algebra $\overline{C}$, let

$$\mathbb{P}_0(\overline{C}(\mathcal{H})) \stackrel{\text{def}}{=} \sum_{\overline{y} \in \mathcal{H}} p_{LN}(\overline{\boldsymbol{y}}) \tag{22}$$

First, we show that $\mathbb{P}_0$ is a well-defined function (an expression whose definition assigns it a unique interpretation or value.)

Proof. Suppose a cylinder set can be described as two different prefix sets: $H_1 \subseteq \overline{\Sigma}^{k_1}$ and $H_2 \subseteq \overline{\Sigma}^{k_2}$. In other words, $\overline{C}(H_1) = \overline{C}(H_2)$.[3] Without loss of generality, assume that $k_1 \leq k_2$. Then, [4]

[3] why?

[4] Typos in course note, missing overline in second $\boldsymbol{y}$ of $\boldsymbol{yy}$

$$\begin{aligned}
\overline{C}(H_2) &= \overline{C}(H_1) && (23)\\
&= \bigcup_{\boldsymbol{y} \in H_1} \overline{C}(\boldsymbol{y}) && (24)\\
&= \bigcup_{\boldsymbol{y} \in H_1} \bigcup_{\overline{\boldsymbol{y}} \in \overline{\Sigma}^{k_2 - k_1}} \overline{C}(\boldsymbol{y}\overline{\boldsymbol{y}}) && (25)
\end{aligned}$$

All unions above are disjoint, and hence $H_2 = \bigcup_{\overline{\boldsymbol{y}} \in \overline{\Sigma}^{k_2 - k_1}} \{\boldsymbol{y}\overline{\boldsymbol{y}} | \boldsymbol{y} \in H_1\}$. Then, by the locally-normalizing property of p_{LN}, we have that

$$\mathbb{P}_0(\overline{C}(H_1)) = \mathbb{P}_0(\overline{C}(H_2)) \tag{26}$$

□

Now, we can prove the following:

Lemma 2.5.2 (script)

$\mathbb{P}_0$ is a pre-measure over $\overline{\mathcal{C}}$

Lemma 2.5.3 (script)

Let $\mathbb{P}_0$ be a finitely additive probability pre-measure over $\overline{C}$, such that, given a decreasing sequence of sets $A_1 \supset A_2 \ldots$ in $\overline{\mathcal{C}}$ where $\bigcap_{n-1}^{\infty} A_n = \varnothing$, $\lim_{n \to \infty} \mathbb{P}_0(A_n) = 0$. Then, $\mathbb{P}_0$ is also countably additive over $\overline{\mathcal{C}}$.

First, we prove Lemma 2.5.3

Proof. Let $\{A_n\}$ be a sequence of disjoint sets in $\overline{C}$ such that $A = \bigcup_n A_n \in \overline{\mathcal{C}}$ Then, defining $B_n = \bigcup_{m > n} A_m$ we see that $B_1 \supset B_2 \supset \ldots$

and $\bigcap_n B_n = \varnothing$. Notice that for any n

$$A = A_1 \cup \cdots \cup A_n \cup B_n \tag{27}$$

and hence by finite additivity of $\mathbb{P}_0$

$$\mathbb{P}_0(A) = \mathbb{P}_0(A_1) + \cdots + \mathbb{P}_0(A_n) + \mathbb{P}_0(B_n) \tag{28}$$

equivalent to

$$\mathbb{P}_0(A_1) + \cdots + \mathbb{P}_0(A_n) = \mathbb{P}_0(A) - \mathbb{P}_0(B_n) \tag{29}$$

Since $B_n \downarrow \varnothing$ implies[5] that $\mathbb{P}_0(B_n) \downarrow 0$ by assumption, taking the limits on both sides of the above equation yields

[5] $B_n \downarrow \varnothing$ means B_n is decreasing and converges to the empty set

$$\sum_n \mathbb{P}_0(A_n) = \lim_{n\to\infty} \sum_{i\leq n} \mathbb{P}_0(A_i) = \mathbb{P}_0(A) - \lim_{n\to\infty} \mathbb{P}_0(B_n) = \mathbb{P}_0(A) \tag{30}$$

which shows countable additivity. □

We can now proof **Lemma 2.5.2**. That $\mathbb{P}_0$ is a pre-measure over $\overline{\mathcal{C}}$.

Proof. First, we show that $\mathbb{P}_0$ is finitely additive over $\overline{\mathcal{C}}$. Let $\mathcal{C}(\mathcal{H}_1)$ and $\mathcal{C}(\mathcal{H}_2)$ be disjoint cylinder sets. Using the fact that $\mathbb{P}_0$ is a well-defined function, we can assume $\mathcal{C}(\mathcal{H}_1)$ and $\mathcal{C}(\mathcal{H}_2)$ are the same rank without loss of generality. Then, by definition of cylinder sets, we have:

$$\mathcal{C}(\mathcal{H}_1) \cup \mathcal{C}(\mathcal{H}_2) = \left\{ \boldsymbol{y}\boldsymbol{\omega} | \boldsymbol{y} \in \mathcal{H}_1, \boldsymbol{\omega} \in \overline{\Sigma}^\infty \right\} \cup \left\{ \boldsymbol{y}\boldsymbol{\omega} | \boldsymbol{y} \in \mathcal{H}_2, \boldsymbol{\omega} \in \overline{\Sigma}^\infty \right\} \tag{31}$$

$$= \left\{ \boldsymbol{y}\boldsymbol{\omega} | \boldsymbol{y} \in \mathcal{H}_1 \cup \mathcal{H}_2, \boldsymbol{\omega} \in \overline{\Sigma}^\infty \right\} \tag{32}$$

$$= \mathcal{C}(\mathcal{H}_1 \cup \mathcal{H}_2) \tag{33}$$

Where we used the fact $\mathcal{H}_1$, $\mathcal{H}_2$ equal rank, disjoint. This leads to

$$\mathbb{P}_0(\mathcal{C}(\mathcal{H}_1) \cup \mathcal{C}(\mathcal{H}_2)) = \mathbb{P}_0(\mathcal{C}(\mathcal{H}_1 \cup \mathcal{H}_2)) \tag{34}$$

$$= \sum_{\boldsymbol{y} \in \mathcal{H}_1 \cup \mathcal{H}_2} p_{\mathrm{LN}}(\boldsymbol{y}) \tag{35}$$

$$= \sum_{\boldsymbol{y} \in \mathcal{H}_1} p_{\mathrm{LN}}(\boldsymbol{y}) + \sum_{\boldsymbol{y} \in \mathcal{H}_2} p_{\mathrm{LN}}(\boldsymbol{y}) \tag{36}$$

$$= \mathbb{P}_0(\mathcal{C}(\mathcal{H}_1)) + \mathbb{P}_0(\mathcal{C}(\mathcal{H}_2)) \tag{37}$$

Hence, $\mathbb{P}_0$ is finitely additive over $\overline{\mathcal{C}}$.

Now we show $\mathbb{P}_0$ is countably additive. Equip $\overline{\Sigma}$ with the discrete topology. Since $\overline{\Sigma}$ is finite, so is $\overline{\Sigma}^\infty$ by Tychonoff. Then by properties of the product topology over discrete finite spaces, all cylinder sets in $\overline{\Sigma}^\infty$ are compact. Let $\mathcal{C}_1 \supset \mathcal{C}_2 \supset \ldots$ be a decreasing sequence

of cylinder sets and $\mathcal{C}_1 \cap \mathcal{C}_2 \cap \cdots = \varnothing$. For the sake of contradiction, assume that $\mathbb{P}_0(\bigcap_n \mathcal{C}_n) > 0$, this implies all $\mathcal{C}_n \neq \varnothing$. However, by **Cantors Intersection Theorem**, $\bigcap_n \mathcal{C}_n \neq \varnothing$, contradicting the assumption, hence $\mathbb{P}_0(\bigcap_n \mathcal{C}_n) = 0$, and by Lemma 2.5.3, $\mathbb{P}_0$ is countably additive.

It remains to show $\mathbb{P}_0(\Omega) = 1$. Recall that we have $\Omega = \overline{\Sigma}^\infty = \overline{C}(\overline{\Sigma}^k) \in \overline{\mathcal{C}} = \mathcal{F}$. This means we have

$$\begin{aligned} \mathbb{P}_0(\Omega) &= \mathbb{P}_0(\overline{C}(\overline{\Sigma}^1)) && (38)\\ &= \sum_{\overline{y} \in \overline{\Sigma}} (\overline{y}) && (39)\\ &= \sum_{\overline{y} \in \overline{\Sigma}} p_{\text{LN}}(\overline{y} | \text{BOS}) = 1 && (40) \end{aligned}$$

□

Step 4: Apply Carathéodory's Extension Theorem to extend $\mathbb{P}_0$ into a Measure $\mathbb{P}$

Carathéodory's Extension Theorem

Given an algebra $\mathcal{A}$ over some set Ω and a probability premeasure $\mathbb{P}_0 : \mathcal{A} \to [0, 1]$, there exists a probability space $(\Omega, \mathcal{F}, \mathbb{P})$ such that $\mathcal{A} \subset \mathcal{F}$ and $\mathbb{P}|_{\mathcal{A}} = \mathbb{P}_0$. Furthermore, the σ-algebra $\mathcal{F}$ depends only on $\mathcal{A}$ and is minimal and unique, which is denoted as $\sigma(\mathcal{A})$, and the probability measure $\mathcal{P}$ is unique.

Proof. Chapter 11 in Billingsley (1995), beyond scope of this course. □

Applying Carathéodory's Extension Theorem, we know there exists a probability space $(\overline{\Sigma}^\infty, \sigma(\overline{\mathcal{C}}), \mathbb{P})$.

Step 5: Constructing a R.V (Defining a SM) We now want to construct a σ-algebra over $\Sigma^* \cup \Sigma^\infty$ and then map elements from $\overline{\Sigma}^\infty$ to $\Sigma^* \cup \Sigma^\infty$. Given $\mathcal{H} \subset \Sigma^k$, define a rank-$k$ cylinder set in $\Sigma^* \cup \Sigma^\infty$ to be

$$\mathcal{C}(\mathcal{H}) \stackrel{\text{def}}{=} \{\boldsymbol{y}\boldsymbol{\omega} | \boldsymbol{y} \in \mathcal{H}, \boldsymbol{\omega} \in \Sigma^* \cup \Sigma^\infty\} \tag{41}$$

The suffix $\boldsymbol{\omega}$ comes from $\Sigma^* \cup \Sigma^\infty$, so it means that they can be **finite** and **do not contain eos**. Define $\mathcal{C} \stackrel{\text{def}}{=} \bigcup_{k=1}^\infty \mathcal{C}_k$, then $\sigma(\mathcal{C})$ is a σ-algebra as Carathéodory's Extension Theorem. We can then define the random variable $\boldsymbol{\omega} \in \overline{\Sigma}^\infty$

$$x(\boldsymbol{\omega}) = \begin{cases} \boldsymbol{\omega}_{<k} & \text{if } k \text{ is the fist } \text{EOS} \text{in } \boldsymbol{\omega} \\ \boldsymbol{\omega} & \text{otherwise} \end{cases} \tag{42}$$

A sufficient condition for tightness

An LNM is tight if and only if $\tilde{p}_{\text{EOS}}(t) = 1$ for some t or $\sum_{t=1}^{\infty} \tilde{p}_{\text{EOS}}(t) = \infty$

2 *Classical Language Models (Finite-State Language Models)*

Informal definition of a finite-state language model

A language model p_{LM} is a **finite-state** if it defines only finitely many unique conditional distributions $p_{\text{LM}}(y|\boldsymbol{y})$. In other words, there are only finitely many contexts $\boldsymbol{y}$ which define the distribution over the next symbol $p_{\text{LM}}(y|\boldsymbol{y})$.

2.1 *Weighted Finite-state Automata*

Before we introduce finite-state *language models*, we go into the theory of finite-state *automata*.

Finite-state Automata

A **finite-state automaton** (FSA) is a 5-tuple $(\Sigma, Q, I, F, \delta)$ where

- Σ is an alphabet;
- Q is a finite set of states;
- $I \subseteq Q$ is the set of initial states;
- $F \subseteq$ is the set of final or accepting states;
- A finite multiset $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$. Elements of δ are generally called transitions.

A natural question to ask is what will happen for a state-symbol pair (q, a) when there is *more than one* possible transition allowed under the relation δ. In such case, we take *all* implicit transitions simultaneously, which leads us to a pair of definitions.

Deterministic finite-state automaton

A FSA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is **deterministic** if

- it does not have any ε-transitions;
- for every $(q, a) \in Q \times \Sigma$, there is at most one $q' \in Q$ such that $q \xrightarrow{a} q' \in \delta$;
- there is a single initial state $|I| = 1$.

Otherwise, $\mathcal{A}$ is **non-deterministic**

An important but not obvious result is that the classes of deterministic and non-deterministic FSA are equivalent, in the sense that you can always represent a member of one class with a member of the other. If the automaton ends up, after reading in the last symbol of the input string in one of the final states $q_\psi \in F$, we say that the automaton **accepts** that string. A finite-state automaton is therefore a computational device that determines whether a string satisfies a con-

dition. A string that satisfies this condition is said to be recognized by the automaton and the set of all string satisfying this condition form the *language* of the automaton.

Language of a finite-state automaton

Let $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ be an FSA. The **language** of $\mathcal{A}$, $L(\mathcal{A})$ is defined as

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{\boldsymbol{y} | \boldsymbol{y} \text{ is recognized by } \mathcal{A}\} \tag{43}$$

Abstractly, a FSA is hence a specification of a set of *rules* that strings must satisfy to be included in its language. The set of languages that FSA can recognize is known as the class of **regular languages**.

Regular Language

A language $L \subseteq \Sigma^*$ is **regular** if and only if it can be recognized by an unweighted FSA, i.e., if there exists a FSA $\mathcal{A}$ such that $L = L(\mathcal{A})$.

A common and very useful augmentation to FSA is through the addition of *weights* on the transitions. Usually, the general theory of WFSA makes use of semiring theory (covered in NLP and AFLT), for this course, we focus on real-valued weights.

Real-weighted finite-state automaton

A **real-weighted finite-state automaton** (WFSA) $\mathcal{A}$ is a 5-tuple $(\Sigma, Q, \delta, \lambda, \rho)$ where

- Σ is a finite alphabet;
- Q is a finite set of states;
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \mathbb{R} \times Q$ a finite multiset of transitions;
- $\lambda : Q \to \mathbb{R}$ a weighting function over Q (initial weights);
- $\rho : Q \to \mathbb{R}$ a weighting function over Q (final weights).

So basically, the set of initial states are $I = \{q \in Q | \lambda(q) \neq 0\}$ and final states $F = \{q \in Q | \rho(q) \neq 0\}$.

Of course, similar to graph theory, we can define a WFSA using a transition matrix.

Transition Matrix

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. For any $a \in |\Sigma$, we define the **symbol-specific transition matrx** $\mathbf{T}^{(a)}$ as the transition matrix of the graph restricted to a-labelled transitions. We also define the

(full) **transition matrix** as $\mathbf{T} \stackrel{\text{def}}{=} \sum_{a \in \Sigma} \mathbf{T}^{(a)}$.

A path is an important concept when talking about (weighted) finite-state automata, as it defines the basic structure by which a string is recognized or weighted. We now give a formal definition of a path and discuss how to weight paths.

Path

A **path** $\boldsymbol{\pi}$ is an element of δ^* with *consecutive transitions*. The **length** of a path is the number of transitions in it denoted as $|\pi|$, we use $p(\boldsymbol{\pi})$ and $n(\boldsymbol{\pi})$ to denote the origin and the destination of a path, respectively. The **yield** of the path is the concatenation of the input symbols on the edges along the path, which we will mark with $s(\boldsymbol{\pi})$. We denote the set of paths with Π.

- $\Pi(\mathcal{A})$ as the set of all paths in automaton $\mathcal{A}$;
- $\Pi(\mathcal{A}, \boldsymbol{y})$ as the set of all paths in the automaton $\mathcal{A}$ with the yield $\boldsymbol{y} \in \Sigma^*$;
- $\Pi(\mathcal{A}, q, q')$ as the set of all paths in automaton $\mathcal{A}$ from state q to state q'.

Path Weight

The **inner path weight** $w_I(\boldsymbol{\pi})$ of a path $\boldsymbol{\pi}$ is defined as

$$w_I(\boldsymbol{\pi}) = \prod_{n=1}^{N} w_n \tag{44}$$

The **(full) path weight** of a path $\boldsymbol{\pi}$ is then defined as

$$w(\boldsymbol{\pi}) = \lambda(p(\boldsymbol{\pi}))w_I(\boldsymbol{\pi})\rho(n(\boldsymbol{\pi})) \tag{45}$$

A path $\boldsymbol{\pi}$ is called **accepting** or **successful** if $w(\boldsymbol{\pi}) \neq 0$.

String Acceptance Weights and Weighted Regular Languages When we introduced unweighted FSA, we defined the important concept of recognizing a string and recognizing a language. We generalize these concepts to the very natural quantity of the weight assigned by a WFSA to a string $\boldsymbol{y} \in \Sigma^*$, i.e., its acceptance weight, or stringsum, as the sum of the weights of the paths that yield $\boldsymbol{y}$.

Stringsum

The **stringsum**, string weight or acceptance weight of a string $\boldsymbol{y} \in \Sigma^*$ under a WFSA $\mathcal{A}$ is defined as

$$\mathcal{A}(\boldsymbol{y}) \stackrel{\text{def}}{=} \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A}, \boldsymbol{y})} w(\boldsymbol{\pi}) \tag{46}$$

This naturally generalizes the notion of acceptance by an unweighted FSA -whereas an unweighted FSA only makes a binary decision of accepting or rejecting a string, a WFSA always accepts a string with a specific weight. This leads to the definition of weighted language of the WFSA.

Weighted language of a WFSA

Let $\mathcal{A}$ be a WFSA. Its **(weighted) language** is defined as

$$L(\mathcal{A}) \stackrel{\text{def}}{=} \{(\boldsymbol{y}, \mathcal{A}(\boldsymbol{y})) | \boldsymbol{y} \in \Sigma^*\} \tag{47}$$

We say that a language is a weighted regular language if it is a language of some WFSA. Lastly, we also define the full and state-specific allsum of the automaton. The former refers to the total weight assigned to *all* possible strings, or all possible paths where the latter refers to the sum of path weights of the paths stemming from a specific state.

State-specific allsum

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. The **allsum** of a state $q \in Q$ is defined as

$$Z(\mathcal{A}, q) = \sum_{\boldsymbol{\pi} \in \Pi(\mathcal{A}), q_1 = q} w_I(\boldsymbol{\pi}) \rho(n(\boldsymbol{\pi})) \tag{48}$$

State-specific allsums are also referred to as **backward values** and are often denoted as $\beta(q)$.

WFSA allsum

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. The **allsum** of $\mathcal{A}$ is defined as

$$Z(\mathcal{A}) = \sum_{y \in \Sigma^*} \mathcal{A}(\boldsymbol{y}) = \sum_{y \in \Sigma^*} \sum_{\pi \in \Pi(\mathcal{A}, \boldsymbol{y})} w(\boldsymbol{\pi}) = \sum_{\boldsymbol{\pi} = \Pi(\mathcal{A})} w(\boldsymbol{\pi}) \tag{49}$$

Accessibility and Probabilistic WFSA An important property of states of a WFSA which we will need when investigating the tightness of finite-state language models is accessibility.

(Co)-Accessible and useful states

A state $q \in Q$ of a WFSA is **accessible** if there is a non-zero-weighted path to q from some state q' with $\lambda(q') \neq 0$; it is **co-accessible state** if there is a non-zero-weighted path from q to some state q'' with $\rho(q'') \neq 0$. It is **useful** if it is both accessible and co-accessible, i.e., q appears on some non-zero-weighted accepting path.

Trim automaton

Trimming a WFSA means removing its useless states. Removing the non-useful states means removing their rows and columns from $\mathbf{T}$ as well as their rows from $\overrightarrow{\lambda}$ and $\overrightarrow{\rho}$, yielding possibly smaller $\mathbf{T}'$, $\overrightarrow{\lambda}'$ and $\overrightarrow{p}'$.

We will use WFSAs to specify language models. However, not every WFSA is a language model. Generally, the weight of a string could be negative if we allow arbitrary real weights. Thus, a restriction we will impose on all weighted automata that represent finite-state language models is that the weights be non-negative. Furthermore, a special class of WFSAs that will be particular interest later is probabilistic WFSAs.

Probabilistic Weighted Finite-State Automaton

A WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ is a **probabilistic** FSA (PFSA) if

$$\sum_{q \in Q} \lambda(q) = 1 \tag{50}$$

and, for all $q \in Q$ and all outgoing transitions $q \xrightarrow{a/w} q' \in \delta$ it holds that $\lambda(q) \geq 0$, $\rho(q) \geq 0$, $w \geq 0$ and

$$\sum_{q \xrightarrow{a/w} q} w + \rho(q) = 1 \tag{51}$$

This means that the initial weights of all the states of the automaton form a probability distribution (the initial weight of a state corresponds to the probability of starting in it), as well as that, for any state q in the WFSA, the weights of its outgoing transitions (with any label) together with its final weight form a valid discrete probability

distribution. In a certain way, probabilistic FSA naturally correspond to locally normalized language models.

2.2 *Finite-state Language Models*

We can now formally define what it means for a language model to be finite-state:

Finite-state language models

A language model p_{LM} is **finite-state** if it can be represented by a weighted finite-state automaton, i.e., if there exists a WFSA $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ such that $L(\mathcal{A}) = L(p_{\text{LM}})$.

Given an WFSA $\mathcal{A}$, there are two established ways of defining a *probability* of string. In a probabilistic FSA, any action from a state $q \in Q$ is associated with a probability. Since the current state completely encodes all the information of the input seen so far in a finite-state automaton, it is intuitive to see those probabilities as conditional probabilities of the next symbol given the input seen so far. One can, therefore define the probability of a path as the product of these individual "conditional" probabilities.

String probabilities in a PFSA.

Path Probability in a PFSA

We call the weight of a path $\boldsymbol{\pi} \in \Pi(\mathcal{A})$ in a probabilistic FSA the **probability** of the path $\boldsymbol{\pi}$.

This alone is however not enough to define the probability of any particular string $\boldsymbol{y} \in \Sigma^*$ since there might be multiple accepting paths for $\boldsymbol{y}$. Naturally, we define the probability of $\boldsymbol{y}$ as the sum of the individual paths that recognize it:

String probability in a PFSA

We call the stringsum of a string $\boldsymbol{y} \in \Sigma^*$ in a PFSA the **probability** of the string $\boldsymbol{y}$

$$p_{\mathcal{A}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \mathcal{A}(\boldsymbol{y}) \tag{52}$$

Notice that these two definitions did not require any *normalization* over all possible paths or strings, this closely resembles the way we defined a locally normalized models based on the conditional probabilities of a sequence model. Such definitions of string probabilities are

attractive as the summation over all possible strings is avoided. **Are they tight however? Do they sum to 1?**

String Probabilities in a General WFSA To define string probabilities in a general WFSA, we used the introduced notations of stringsum and the allsum. The allsum allows us to tractably *normalize* the stringsum to define the globally normalized probability of a string $\boldsymbol{y}$ as the proportion of the total weight assigned to all strings that is assigned to $\boldsymbol{y}$.

String probability in a WFSA

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a normalizable WFSA with non-negative weights. We define the probability of a string $\boldsymbol{y} \in \Sigma^*$ under $\mathcal{A}$ as

$$p_{\mathcal{A}}(\boldsymbol{y}) \stackrel{\text{def}}{=} \frac{\mathcal{A}(\boldsymbol{y})}{Z(\mathcal{A})} \tag{53}$$

A language model induced by a WFSA

Let $\mathcal{A} = (\Sigma, Q, \delta, \lambda, \rho)$ be a WFSA. We define the **language model induced by** $\mathcal{A}$ as the following probability distribution over Σ^*

$$p_{\mathrm{LM}\mathcal{A}}(\boldsymbol{y}) \stackrel{\text{def}}{=} p_{\mathcal{A}}(\boldsymbol{y}) \tag{54}$$

It is easy to see that while global normalization requires the computation of the allsum, language models induced by WFSAs, through our definition are *globally normalized*, and thus always tight. In the next section, we will consider how the quantities for needing our definition can be computed. Particularly, $Z(\mathcal{A})$, as this involves the summation over possibly infinitely many terms and therefore requires some clever tricks.

2.3 *Normalizing Finite-state Language Models*

In this subsection, we develop an algorithm for normalizing a globally normalized language model defined by a WFSA, an algorithm for computing the all sum $Z(\mathcal{A})$ whenever this quantity is finite. Moreover, this derivation will also reveal necessary and sufficient conditions for WFSAs to be normalizable.

Converting a matrix of pairwise pathsums to the allsum. Before we consider how to compute $Z(\mathcal{A})$, let us consider a simpler problem. Suppose we had a matrix $\mathbf{M}$, which contained at the entry $\mathbf{M}_{ij}$ the sum

of all inner weights over all paths between the states i and j, i.e.,

$$M_{ij} = \sum_{\pi \in \Pi(\mathcal{A}, i, j)} w_I(\pi) \tag{55}$$

How could we compute $Z(\mathcal{A})$?

$$\begin{aligned}
Z(\mathcal{A}) &= \sum_{\pi \in \Pi(\mathcal{A})} w(\pi) && (56)\\
&= \sum_{\pi \in \Pi(\mathcal{A})} \lambda(p(\pi)) w_I(\pi) \rho(n(\pi)) && (57)\\
&= \sum_{i,j \in Q} \sum_{\pi \in \Pi(\mathcal{A}, i, j)} \lambda(p(\pi)) w_I(\pi) \rho(n(\pi)) && (58)\\
&= \sum_{i,j \in Q} \lambda(p(\pi)) \Big(\sum_{\pi \in \Pi(\mathcal{A}, i, j)} w_I(\pi) \Big) \rho(n(\pi)) && (59)\\
&= \sum_{i,j \in Q} \lambda(i) \mathbf{M}_{i,j} \rho(j) && (60)\\
&= \overrightarrow{\lambda} \mathbf{M} \overrightarrow{\rho} && (61)
\end{aligned}$$

Computing the matrix of pairwise pathsums Let $\mathbf{T}$ be the transition matrix of the automaton $\mathcal{A}$. Notice that the entry $\mathcal{T}_{ij}$ by definition contains the sum of the inner weights of all paths of length exactly 1 (individual transitions) between the states i and j. We also define $\mathbf{T}^0 = \mathbf{I}$, meaning that the sum of the weights of the paths between i and j of length zero is 0 if $i \neq j$ and q. If $i = j$ this corresponds to not transitioning, i.e., staying in place, if $i = j$. We state a basic result from graph theory.

Let $\mathbf{T}$ be the transition matrix of some weighted directed graph $\mathcal{G}$. Then the matrix $\mathbf{T}^d$ contains the allsum of paths of length *exactly* d, i.e.

$$\mathbf{T}^d_{i,j} = \sum_{\pi \in \Pi(\mathcal{A}, i, j), |\pi| = d} w_I(\pi) \tag{62}$$

Proof. The proof was done in NLP Assignment 3. □

Then, it follows directly that the matrix

$$\mathbf{T}^{\leq d} \stackrel{\text{def}}{=} \sum_{k=1}^{d} \mathbf{T}^k \tag{63}$$

contains the pairwise pathsums of paths of length *at most* d. In general, the WFSA representing a *n-gram* language model can of course be cyclic. THis means that the number of paths in $\Pi(\mathcal{A})$ might be infinite and they might be arbitrary length (which is the result of looping in a cycle arbitrarily many times). To compute the pairwise pathsums over *all* possible paths, we, therefore have to compute

$$\mathbf{T}^* \stackrel{\text{def}}{=} \lim_{d \to \infty} \mathbf{T}^{\leq d} = \sum_{d=0}^{\infty} \mathbf{T}^d \tag{64}$$

This is exactly the matrix form of the geometric sum, similarly to the scalar, we can manipulate the above equation to arrive to a closed-form expression for computing it.

$$\mathbf{T}^* = \sum_{d=0}^{\infty} \mathbf{T}^d \tag{65}$$

$$= \mathbf{I} + \sum_{d=1}^{\infty} \mathbf{T}^d \tag{66}$$

$$= \mathbf{I} + \sum_{d=1}^{\infty} \mathbf{T}\mathbf{T}^{d-1} \tag{67}$$

$$= \mathbf{I} + \mathbf{T} \sum_{d=1}^{\infty} \mathbf{T}^{d-1} \tag{68}$$

$$= \mathbf{I} + \mathbf{T} \sum_{d=0}^{\infty} \mathbf{T}^{d} \tag{69}$$

$$= \mathbf{I} + \mathbf{T}\mathbf{T}^* \tag{70}$$

If the inverse of $\mathbf{I} - \mathbf{T}$ exists, we can further rearrange this equation to arrive at

$$\mathbf{T}^* = \mathbf{I} + \mathbf{T}\mathbf{T}^* \tag{71}$$

$$\mathbf{T}^* - \mathbf{T}\mathbf{T}^* = \mathbf{I} \tag{72}$$

$$\mathbf{T}^* - \mathbf{T}^*\mathbf{T} = \mathbf{I} \tag{73}$$

$$\mathbf{T}^*(\mathbf{I} - \mathbf{T}) = \mathbf{I} \tag{74}$$

$$\mathbf{T}^* = (\mathbf{I} - \mathbf{T})^{-1} \tag{75}$$

This means that, if $\mathbf{I} - \mathbf{T}$ exists, we can compute the pairwise pathsums by simply inverting it! Using the remark above on how to convert a matrix of pairwise pathsums into the full allsum, we can therefore see that we can globally normalize an n-gram language model via matrix inversion! The runtime of inverting a $N \times N$ matrix is $\mathcal{O}(N^3)$, and $N = |Q|$ for a transition matrix of a WFSA with states Q, we can globally normalize n-gram language model in time cubic in the number of its states. This is a special case of Lehmann (1977). We still have to determine when the infinite sum in Equation 64 converges, one can see by writing the product $\mathbf{T}^d$ in terms of its eigenvalues that the entries of $\mathbf{T}^d$ diverge towards $\pm\infty$ as soon as the magnitude of any of $\mathbf{T}$'s eigenvalues are larger than 1. This means $\|T\|_2 < 1$ (spectral norm) is a necessary condition for the infinite sum to exist. This is, however, also a *sufficient* condition: if $\|T\|_2 < 1$, all of $\mathbf{T}$'s eigenvalues are smaller than 1 in magnitude, meaning the eigenvalues of $\mathbf{I} - \mathbf{T}$ are strictly positive and the matrix $\mathbf{I} - \mathbf{T}$ is invertible. We can speed this up by decomposing the automaton into SCCs (see AFLT Assignment 3), if it decomposes.

Locally Normalizing a Globally Normalized Finite-state Language Model As **any LM** (and thus any **globally-normalized**) model with a normalizable energy function can also be locally normalized. In the case of finite-state language models, we can actually explicitly construct the WFSA representing the locally normalized variant using a procedure that is conceptually similar to the allsum algorithm described here. In contrast to the procedure presented here, the local normalization algorithm computes the pathsums of the paths stemming from every possible state q indiviudally and then reweights the transitions depending on the pathsums of their target states r. This is an instance of the more general **weight pushing** algorithm.

PFSAs and WFSAs are equally expressive

Normalizable weighted finite-state automata with non-negative weights and tight probabilistic finite-state automata are equally expressive.

2.4 *Tightness of Finite-state Models*

Any normalizable globally normalized finite-state language model is tight by definition because the sum of the scores over all finite strings is finite, and since they are normalized, they sum to 1. We therefore focus on locally normalized finite-state models and provide necessary and sufficient conditions for their tightness. Locally normalized finite-state models are exactly PFSAs, the tightness of PWFSA can be easily characterized as

A sufficient condition for tightness of finite-state language models

A PFSA is tight if and only if all accessible states are also co-accessible.

2.5 *The n-gram assumption*

***n*-gram**

assumption In words, the n-gram assumption states the probability of a word y_t only depends on $n-1$ previous words $y_{t-1}, \ldots y_{t-n+1}$ where $y_0 \stackrel{\text{def}}{=} \text{BOS}$. We can write the n-gram assumption as a conditional independence assumption, i.e,

$$p_{\text{SM}}(y_t|\boldsymbol{y}_{<t}) \stackrel{\text{def}}{=} p_{\text{SM}}(y_t|y_{t-1}\ldots y_{t-n+1}) \stackrel{\text{def}}{=} p_{\text{SM}}(y_t|\boldsymbol{y}_{t-n-1:t-1}) \tag{76}$$

The sequence $y_{t-1} \dots y_{t-n+1}$ is often called the **history** or the **context**.

Given this definition, where the conditional probabilities $p_{\text{SM}}(y_t|\boldsymbol{y}_{t-n-1:t-1})$ depends on $n-1$ previous symbols, we can run into problems with negative indices for $t < n$. To handle these cases, we **pad** the sequences with the BOS symbols at the beginning, that is, we will assume the sequence $\bar{y}_1 \dots \bar{y}_t$ for $t < n-1$ are transformed as

$$\bar{y}_1 \bar{y}_2 \dots \bar{y}_t \mapsto \underbrace{\text{BOS} \dots \text{BOS}}_{n-1-t\text{times}} \bar{y}_1 \bar{y}_2 \dots \bar{y}_t \tag{77}$$

By definition, n-gram language models can only model dependencies spanning n tokens or less, by limiting the length of the relevant context when determining $p_{SM}(y_t|\boldsymbol{y}_{<t})$ to the previous n tokens, the n-gram assumption limits the number of possible probability distributions that is needed to be tracked to $\mathcal{O}(|\Sigma|^{n-1})$. This is still huge, the next section deals with this.

2.6 *Representation-based n-gram Models*

In this section, we consider for the first time what an actual implementation of a finite-state, or more precisely, a n-gram language model might look like. Concretely, we will define our first parameterized language model in our General language modeling framework by defining a particular form of the encoding function ENC as a simple multilayer feed-forward neural network.

Let us first consider an alternative, possibly the simplest way to define a (locally normalized) n-gram language model: by directly parametrizing the probabilities of each of the symbols y in the distribution $p_{\text{SM}}(\bar{y}|\bar{\boldsymbol{y}})$ for any context $\bar{\boldsymbol{y}}$, that is

$$\boldsymbol{\theta} \stackrel{\text{def}}{=} \left\{ \theta_{y|\boldsymbol{y}} \stackrel{\text{def}}{=} p_{\text{SM}}(y|\boldsymbol{y}) \;\middle|\; y \in \bar{\Sigma}, \boldsymbol{y} \in \bar{\Sigma}^{n-1}, \theta_{y|\boldsymbol{y}} \geq 0, \sum_{y' \in \bar{\Sigma}} \theta_{y'|\boldsymbol{y}} = 1 \right\} \tag{78}$$

The MLE solution is therefore

$$p_{\text{SM}}(y_n|\boldsymbol{y}_{<n}) = \frac{C(y_1, \dots, y_n)}{C(y_1, \dots, y_{n-1})} \tag{79}$$

The model defined above is however, unable to take into account the relationships and similarities between words. The general modeling framework [6] allows us to remedy this using the **distributed word representations**.

To be able to use the embeddings in our general framework, we now just have to define the concrete form of the context-encoding function

[6] In that framework, we associate each word y with its vector representation $\mathbf{e}(y)$, i.e. its embedding, and we combine those into the embedding matrix $\mathbf{E}$

ENC. In the case of the neural n-gram model, which we consider here and as defined by (Bengio et el., 2003), the representations of the context $\boldsymbol{y}_{<t}$, $\text{ENC}(\boldsymbol{y}_{<t})$ are defined as the output of a neural network which looks at the previous $n-1$ words in the context:

$$\text{ENC}(\boldsymbol{y}_{<t}) \stackrel{\text{def}}{=} \text{ENC}(y_{t-1}, y_{t-2}, \ldots, y_{t-n+1}) \tag{80}$$

where ENC is a neural network we define in more detail shortly. The full language model is therefore defined through the conditional distributions

$$p_{\mathbf{SM}}(\bar{y}_t | \bar{\boldsymbol{y}}_{<t}) \stackrel{\textbf{def}}{=} \text{softmax}\Big(\text{ENC}(y_{t-1}, y_{t-2}, \ldots, y_{t-n+1})^\top \mathbf{E} + \mathbf{b}\Big)_{\bar{y}_t} \tag{81}$$

resulting in the locally normalized model

$$p_{\text{LN}}(\boldsymbol{y}) = \text{softmax}\Big(\text{ENC}(\bar{y}_T, \bar{y}_{T-1}, \ldots, \bar{y}_{T-n+2})^\top \mathbf{E} + \mathbf{b}\Big)_{\text{EOS}} \cdot \prod_{t=1}^{T} \text{softmax}\Big(\text{ENC}(y_{t-1}, y_{t-2}, \ldots, y_{t-n+1})^\top \mathbf{E} + \mathbf{b}\Big)_{\bar{y}_t} \tag{82}$$

for $\boldsymbol{y} \in \Sigma^*$.

3 *Recurrent Neural Language Models*

Recurrent neural networks capture the idea of *sequential* processing of strings relatively naturally while also making decisions based on an *infinite* context. They are a class of models that is theoretically capable of recognizing all computatble languages.

Real-valued Recurrent Neural Network

Let Σ be an alphabet. A (deterministic) **real-valued recurrent neural network** $\mathcal{R}$ is a four-tuple $(\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ where

- Σ is the alphabet of input symbols;
- D is the dimension of $\mathcal{R}$;
- $\mathbf{f} : \mathbb{R}^D \times \Sigma \to \mathbb{R}^D$ is the dynamics map, i.e., a function defining the transitions between subsequent states;
- $\mathbf{h}_0 \in \mathbb{R}^D$ is the **initial state**.

We can analogously define **rational-valued recurrent neural networks** as RNNs with the hidden state space $\mathbb{Q}^D$ instead of $\mathbb{R}^D$.

To define *language models* using recurrent neural networks, we will use them as encoder functions $\textsc{enc}$ in our general language modeling framework. To connect the previous language model with the general LM framework, we define the RNN encoding function.

Recurrent Neural Encoding Function

Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be a RNN. A **recurrent neural encoding function** $\textsc{enc}_{\mathcal{R}}$ is a representation function that recursively encodes strings of arbitrary lengths using its dynamics map $\mathbf{f}$.

$$\textsc{enc}_{\mathcal{R}}(\boldsymbol{y}_{<t+1}) \stackrel{\text{def}}{=} \mathbf{f}(\textsc{enc}_{\mathcal{R}}(\boldsymbol{y}_{<t}), y_t) \in \mathbb{R}^D \tag{83}$$

and

$$\textsc{enc}_{\mathcal{R}}(\boldsymbol{y}_{<1}) \stackrel{\text{DEF}}{=} \mathbf{h}_0 \in \mathbb{R}^D \tag{84}$$

Hidden State

Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be an RNN. The hidden state $\mathbf{h}_t \in \mathbb{R}^D$ describes the state of $\mathcal{R}$ after reading y_t. It is recursively computed according to the dynamics map $\mathbf{f}$ as follows:

$$\mathbf{h}_t \stackrel{\text{def}}{=} \textsc{enc}_{\mathcal{R}}(\boldsymbol{y}_{t<t+1}) = \mathbf{f}(\mathbf{h}_{t-1}, y_t) \tag{85}$$

Recurrent Neural Sequence Models Note that a RNN based on the previous definitions on its own does not yet defined a sequence model,

but simply a *context encoding function* $\textsc{enc}_{\mathcal{R}} : \bar{\Sigma}^* \to \mathbb{R}^D$. To define a sequence model based on an RNN, we simply plug the RNN encoding function into the general language modeling framework.

Recurrent neural sequence model

Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be a recurrent neural network and $\mathbf{E} \in \mathbb{R}^{|\bar{\Sigma}| \times D}$ a symbol representation matrix. A D-dimensional **recurrent neural sequence model** over an alphabet Σ is a tuple $(\Sigma, D, \mathbf{f}, \mathbf{E}, \mathbf{h}_0)$ defining the sequence model of the form

$$p_{\mathbf{SM}}(y_t | \boldsymbol{y}_{<t}) \stackrel{\text{def}}{=} \mathbf{f}_{\Delta^{|\bar{\Sigma}|-1}}(\mathbf{E}\textsc{enc}_{\mathcal{R}}(\boldsymbol{y}_{<t}))_{y_t} = \mathbf{f}_{\Delta^{|\bar{\Sigma}|-1}}(\mathbf{E}\mathbf{h}_{t-1})_{y_t} \quad (86)$$

By far the most common choice of the projection function is the softmax. In the course notes, they will refer to RNN sequence models whose next-symbol probability distributions are computed using the softmax function as **softmax RNN sequence models**.

One-hot encoding

Let Σ be an alphabet and $n : \Sigma \to \{1, \ldots, |\Sigma|\}$ a bijection (i.e., an ordering of the alphabet, assigning an index to each symbol in Σ). A **one-hot encoding** $[\![\cdot]\!]$ is a representation function of the symbols in Σ which assigns the symbol $y \in \Sigma$ the $n(y)$th basis vector

$$[\![y]\!] \stackrel{\textbf{def}}{=} \mathbf{d}_{n(y)} \quad (87)$$

where here $\mathbf{d}_n$ is the nth canonical basis vector, i.e., a vector of zeros with a 1 at position n.

Activation function

Let $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ be a recurrent neural network. If the hidden states $\mathbf{h}_t$ f the RNN are computed as

$$\mathbf{h}_t = \sigma(\mathbf{g}(\mathbf{h}_{t-1}, y)) \quad (88)$$

for some function $\mathbf{g} : \mathbb{R}^D \times \Sigma \to \mathbb{R}^D$ and some function $\sigma : \mathbb{R} \to \mathbb{R}$ which is computed *element wise*, which is $\sigma(\mathbf{x})_d = \sigma(x_d)$ for all $d = 1, \ldots D$ and $\mathbf{x} \in \mathbb{R}^D$, we call σ and **activation function**.

General Results on Tightness

Tightness of Recurrent Neural Sequence Model

A softmax recurrent neural sequence model is tight if for all times steps t it holds that

$$s\|\mathbf{h}_t\|_2 \leq \log t \tag{89}$$

where $s \stackrel{\text{def}}{=} \max_{y \in \Sigma} \|\mathbf{e}(y) - \mathbf{e}(\text{EOS})\|_2$

RNNs with bounded dynamics maps are tight

A softmax recurrent neural sequence model $\mathcal{R} = (\Sigma, D, \mathbf{f}, \mathbf{h}_0)$ with a bounded dynamics map $\mathbf{f}$, i.e., with a dynamics map $\mathbf{f}$ such that

$$|\mathbf{f}(\mathbf{x})_d| \leq M \tag{90}$$

for some $M \in \mathbb{R}$ for all $d = 1, \ldots, D$ and all $\mathbf{x} \in \mathbb{R}^D$ is tight.

A special case of the above theorem is RNNs with bounded *activation functions*. Those are tight if the activation function itself is bounded. This implies the standard *sigmoid* and *tanh* activted RNNs are tight. However, this does not hold for RNNs with unbounded activation functions, e.g. the ReLU.

Elman Sequence Model

An **Elman sequence model** $\mathbb{R} - (\Sigma, D, \mathbf{U}, \mathbf{V}, \mathbf{E}, \mathbf{b}_h, \mathbf{h}_0)$ is a D-dimensional recurrent sequence model over an alphabet Σ with the following dynamics map

$$\mathbf{h}_t = \sigma(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{V}\mathbf{e}'(y_t) + \mathbf{b}_h) \tag{91}$$

Here, $\mathbf{e}'\cdot : \Sigma \to \mathbb{R}^R$ is the input symbol **embedding function** which represents each symbol $y \in \Sigma$ as a R-dimension vector and σ is an element-wise non-linearity. $\mathbf{b}_h \in \mathbb{R}^D$, $\mathbf{U} \in \mathbb{R}^{D \times D}$, and $\mathbf{V} \in \mathbb{R}^{D \times R}$.

Heaviside function

The **Heaviside** function is defined as

$$H(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{92}$$

Heaviside Elman Network

A **Heaviside Elman network** (HRNN) is an ELman network with the Heaviside function H as the non-linearity.

4 *Transformers*

The main characteristic of RNNs is the use of a single hidden state $\mathbf{h}_t$ to represent an arbitrary prefix of any string $\mathbf{y}_{<t}$ up to the current time step t. While this allows RNNS to model strings of any length, it also means that arbitrarily long strings must be compressed into this hidden vector of fixed size. Intuitively, this becomes increasingy difficult as the length of the context grows: As the amount of information to be compressed into the hidden state increases with the prefix length, the hidden state may struggle to model the entirety of the preceding context.

The simplest naive way to go about this is to retain the contextual encodings of all prefixes of the string, with this, we avoid the need to summarize the entire context into a single state. Having decided to keep around the encodings of all symbols in the string, let us think about parallelizing the process of encoding a string, e.e., computing $\text{enc}(\boldsymbol{y})$, remember the very general way in which RNNs build a representation of the string $\boldsymbol{y}_{<t}$ by incrementally modifying $\mathbf{h}_t$. The workaround for the issues with sequential processing of RNNs is to process the context for each y_t *independently*, without relying on the encodings of the previous symbols, thus avoiding the sequential bottleneck. Nevertheless, we still want the contextual encoding of y_t to contain information about the rest of the string, i.e., the preceding context. How can we achieve that without relying on recurrence? Again, we grab onto the simplest solution: to compute the symbol encodings for each symbol y_t, based on only the static symbol encodings $\mathbf{e}'(y_t)$, which do not require any recurrence.

4.1 *Formal Definition of Transformers*

Transformer Network

A **transformer network** $\mathcal{T}$ is a tuple $(\Sigma, D, \textsc{enc}_{\mathcal{T}})$ where
- Σ is the alphabet of input symbols,
- D is the dimension of $\mathcal{T}$, and
- $\textsc{enc}_{\mathcal{T}}$ is the transformer encoding function.

Transformer Network

Let $\mathcal{T} = (\Sigma, D, \textsc{enc}_{\mathcal{T}})$ be a transformer network. The hidden state $\mathbf{h}_t \in \mathbb{R}^D$ describes the state of $\mathcal{T}$ after reading $\boldsymbol{y}_{\leq t}$. It is defined with respect to the transformer encoding function $\textsc{enc}_{\mathcal{T}}$

as follows

$$\mathbf{h}_t \stackrel{\text{def}}{=} \textsc{enc}_{\mathcal{T}}(\boldsymbol{y}_{\leq t}) \tag{93}$$

The hidden state $\mathbf{h}_t$ of the transformer does not have any dependence on the preceding hidden states themselves.

Transformer sequence model

Let $\mathcal{T}$ be a transformer network and $\mathbf{E} \in \mathbb{R}^{|\bar{\sigma}| \times D}$ a symbol representation matrix. A D-dimensional **transformer sequence model** over the alphabet Σ is a tuple $(\Sigma, D, \textsc{enc}_{\mathcal{T}}, \mathbf{E})$ defining the sequence model of the form

$$p_{\text{SM}}(\bar{y}_t | \mathbf{y}_{<t}) \stackrel{\text{def}}{=} \text{softmax}(\mathbf{E}\mathbf{h}_{t-1}) = \text{softmax}(\mathbf{E}\textsc{enc}_{\mathcal{T}}(\boldsymbol{y}_{<t}))_{\bar{y}_t} \tag{94}$$

To avoid over-compressing information about sentences into a single vector, a transformer retains the encodings (captured in the hidden states) of all possible prefixes of the string, which we can equivalently simply regard as encodings of individual symbols augmented with the information from the preceding string. However, rather than computing the encoding sequentially like an RNN, the encodings of the individual symbols are computed with so-called attention mechanism.

Attention

Let $f : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ be a **scoring function** and $\mathbf{f}_{\Delta^{D-1}}$ a projection function. Furthermore, let $\mathbf{q} \in \mathbb{R}^D$, $\mathbf{K}_t = (\mathbf{k}_1^\top, \ldots, \mathbf{k}_t^\top) \in \mathbb{R}^{t \times D}$ and $\mathbf{V}_t = (\mathbf{v}_1^\top, \ldots, \mathbf{v}_t^\top) \in \mathbb{R}^{t \times D}$. **Attention** over $\mathbf{K}_t, \mathbf{V}_t$, also denoted by $\texttt{Att}(\mathbf{q}_t, \mathbf{K}_t, \mathbf{V}_t) : \mathbb{R}^D \times \mathbb{R}^{t \times D} \times \mathbb{R}^{t \times D} to \mathbb{R}^D$ is a function computing the vector $\mathbf{a}$ in the following two-step process:

$$\mathbf{s}_t = (s_1, \ldots, s_t) \stackrel{\text{def}}{=} \mathbf{f}_{\Delta^{D-1}}(f(\mathbf{q}, \mathbf{k}_1), f(\mathbf{q}, \mathbf{k}_2), \ldots, f(\mathbf{q}, \mathbf{k}_t)) \tag{95}$$

$$\mathbf{a}_t = \texttt{Att}(\mathbf{q}_t, \mathbf{K}_t, \mathbf{V}_t) \stackrel{\text{def}}{=} s_1\mathbf{v}_1 + s_2\mathbf{v}_2 + \cdots + s_t\mathbf{v}_t \tag{96}$$

The scoring function f is abstractly simply a parameter of the model which we can choose freely, intuitively, it should express the relevance of a particular key $\mathbf{k}$ to the query $\mathbf{q}$, the more the key is relevant to the query, the more attention the model will put the value associated to that key. The projection function then transforms the computed scores ensuring that the transformed scores sum to 1. The vector of the transformed scores $\mathbf{s}$ is then used to compute the result of the attention function - the vector $\mathbf{a}$, which is a convex combination of the

values $\mathbf{v}$ passed to the attention function.

Abstractly, therefore the *keys* contain the information used for <indexing> the *values* with the specific *query*.

The scoring function The scoring function is supposed to measure the <relevance> of a particular value for a query $\mathbf{q}$ through the values' keys. The most common choice for f is the dot product between the query and key, which is often *scaled* by the square root of the vector dimensionalality.

$$f(\mathbf{q}, \mathbf{k}) = \frac{1}{\sqrt{D}} \langle \mathbf{q}, \mathbf{k} \rangle \tag{97}$$

Transformer Layer

Let Q, K, V, and O be parameterized functions from $\mathbb{R}^D$ to $\mathbb{R}^D$. A **transformer layer** is a function $\mathsf{T} : \mathbb{R}^{T \times D} \to \mathbb{R}^{T \times D}$ that takes as input a sequence of vectors $\mathbf{X} = (\mathbf{x}_1^\top, \mathbf{x}_2^\top, \ldots, \mathbf{x}_T^\top)$ and returns $\mathbf{Z} = (\mathbf{z}_1^\top, \mathbf{z}_2^\top, \ldots, \mathbf{z}_T^\top) \in \mathbb{R}^{T \times D}$ according to the following steps:

$$\mathbf{a}_t = \mathtt{Att}(Q(\mathbf{x}_t), K(\mathbf{X}_t), V(\mathbf{X}_t)) + \mathbf{x}_t \tag{98}$$

$$\mathbf{z}_t = O(\mathbf{a}_t) + \mathbf{a}_t \tag{99}$$

for $t = 1, \ldots, T$, so that $\mathsf{T}(\mathbf{X}) \stackrel{\text{def}}{=} \mathbf{Z} = (\mathbf{z}_1^\top, \mathbf{z}_2^\top, \ldots, \mathbf{z}_T^\top) \in \mathbb{R}^{T \times D}$

We now have all the building blocks to define the full transformer architecture, which computes the encodings of the string prefixes (and thus the hidden states).

Transformer

For $L \in \mathbb{N}$, we define a L-layer **transformer** model as a D-dimensional transformer sequence model over an alphabet Σ where the hidden state $\mathbf{h}_t \stackrel{\text{def}}{=} \textsc{enc}_{\mathcal{T}}(y_1, \ldots, y_t) = \textsc{enc}_{\mathcal{T}}(\boldsymbol{y})$ is computed as follows:

$$\mathbf{X}^1 \stackrel{\text{def}}{=} (\mathbf{e}'(y_0), \mathbf{e}'(y_1), \ldots, \mathbf{e}'(y_t)) \tag{100}$$

$$\mathbf{Z}^\ell = \mathsf{T}_\ell(\mathbf{X}^\ell) \quad \text{for } 1 \leq \ell < L \tag{101}$$

$$\mathbf{X}^{\ell+1} = \mathbf{Z}^\ell \quad \text{for } 1 \leq \ell < L \tag{102}$$

$$\mathbf{h}_t = F(\mathbf{z}_t^L) \tag{103}$$

where T_ℓ for $\ell = 1, \ldots, L$ represent L different transformer layers with decoupled parameter. $F : \mathbb{R}^D \to \mathbb{R}^D$ is a transformation

function applied to the contextual encoding of the last symbol in the Lth layer, and $\mathbf{e}' : \Sigma \to \mathbb{R}^D$ is a symbol representation function computing the initial representations of the symbols passed to the first layer of the transformer.

We can now show how the attention mechanism can be conveniently applied to entire strings at once. Specifically, we focus on the case where the attention scoring function f is implemented as a dot-product.

What does the attention mechanism we defined earlier do in this case? Given a query $\mathbf{q}_t$ and a matrix of key values $\mathbf{K} = (\mathbf{k}_1^\top, \ldots, \mathbf{k}_t^\top) \in \mathbb{R}^{t \times D}$, the scoring function simply computes

$$u_j = f(\mathbf{q}_t, \mathbf{k}_j) = \mathbf{q}_t^\top \mathbf{k}_j \tag{104}$$

In this case, the vector $\mathbf{u} = (u_1, \ldots, u_t)$ of *unormalized* attention weights can simply be computed as a single matrix-vector product

$$\mathbf{u} = \mathbf{q}_t^\top \mathbf{K}^\top \tag{105}$$

Furthermore, with this, attention can be easily extended to consider many queries in parallel by stacking multiple queries into a matrix $\mathbf{Q} \stackrel{\text{def}}{=} (\mathbf{q}_1^\top, \mathbf{q}_2^\top, \ldots, \mathbf{q}_t^\top)$, as we detail now. Consider now the product

$$\mathbf{U} = \mathbf{Q}\mathbf{K}^\top \tag{106}$$

Each entry of the resulting matrix U_{ij} is exactly the dot-product between the query $\mathbf{q}_i$ and the key $\mathbf{k}_j$. The rows of $\mathbf{U}$ contain the unnormalized score vectors $\mathbf{u}_i$ from the definition of the attention mechanism. This means if we apply the normalization function $\mathbf{f}_{\Delta^{D-1}}$ *rowwise* (such that the sums of the elements in each row equal 1), we end up with exactly the required normalized scores required for combining the values from the value matrix. We some abuse of notation (just like the paper, which Ryan complained during the NLP lecture), we can simply write that as

$$\mathbf{S} \stackrel{\text{def}}{=} (\mathbf{s}_1^\top, \ldots, \mathbf{s}_t^\top) \stackrel{\text{def}}{=} \mathbf{f}_{\Delta^{D-1}}(\mathbf{U}) = \mathbf{f}_{\Delta^{D-1}}(\mathbf{Q}\mathbf{K}^\top) \tag{107}$$

The rows of $\mathbf{f}_{\Delta^{D-1}}(\mathbf{U})$, therefore, represent the normalized attention weights. This brings us to the final step of the matrix-multiplication-based attention mechanism: Combining the values based on the computed attention weights. Again, this can be performed by a single matrix multiplication. Notice that the *value* vectors are *the same* for all queries - they are simply combined with different attention weights based on the query. Right-multiplying the transposed values matrix

$\mathbf{V} = (\mathbf{v}_1^\top, \ldots, \mathbf{v}_t^\top)$ with $\mathbf{S}$, therefore, perform the convex combination of the value vector $\mathbf{v}_1^\top, \ldots, \mathbf{v}_t^\top$ such that

$$\mathbf{a}_i = \mathbf{s}_i \mathbf{V}^\top = \mathbf{S}_{i,:} \mathbf{V}^\top \tag{108}$$

and thus

$$\mathbf{A} \stackrel{\text{def}}{=} (\mathbf{a}_1, \ldots, \mathbf{a}_t) = \mathbf{S}\mathbf{V}^\top \tag{109}$$

Together, this means that given a sequence of symbol encodings $\mathbf{X}$, we can compute the attention values of all queries with a single matrix multiplication, as long as the scoring function is the (scaled) dot-product. In this course, we refer to this version of attention as an attention block, which intuitively, simply replaces the *element-wise* definition of the attention mechanism defined previously a more efficient and concise definition through matrix multiplications.

Attention Block

Let Q, K, and V be parameterized functions from $\mathbb{R}^{T \times D}$ to $\mathbb{R}^{T \times D}$ and $\mathbf{X} \in \mathbb{R}^{T \times D}$ the matrix of input encodings. An **attention block** is the function $\mathtt{A} : \mathbb{R}^{T \times D} \to \mathbb{R}^{T \times D}$ defined as

$$\mathtt{A}(\mathbf{X}) = \mathbf{f}_{\Delta^{D-1}}\left(Q(\mathbf{X})K(\mathbf{X})^\top\right)V(\mathbf{X}) \tag{110}$$

Further, we define the **attention matrix** as the square matrix $\mathbf{U} \stackrel{\text{def}}{=} Q(\mathbf{X})K(\mathbf{X})^\top \in \mathbb{R}^{T \times T}$.

To recover the autoregressive nature of the language model, we, therefore, posthoc modify the above equation to allow each symbol to attend only to itself and to preceding symbols, while still being able to implement it using matrix multiplication. We do that by adding a mask to zero out the unwanted elements of $\mathbf{U}$.

Masked Attention Block

Let $Q(\cdot), K(\cdot)$, and $V(\cdot)$ be parameterized functions from $\mathbb{R}^{T \times D}$ to $\mathbb{R}^{T \times D}$. A **masked attention block** is a function $\mathtt{A}(\mathbf{X}, \mathbf{M}) : \mathbb{R}^{T \times D} \to \mathbb{R}^{T \times D}$ defined as

$$\mathtt{A}(\mathbf{X}, \mathbf{M}) = \text{softmax}(Q(\mathbf{X})K(\mathbf{X})^\top \odot \mathbf{M})V(\mathbf{X}) \tag{111}$$

where $\odot$ is the element-wise product between matrices, and $\mathbf{M} \in \mathbb{R}^{\ell \times \ell}$, the **masking matrix**, is constructed as follows

$$M_{i,j} = \begin{cases} 1 & \text{if } i \leq j \\ -\infty & \text{otherwise} \end{cases} \qquad \text{for } 0 \leq i, j < T \tag{112}$$

The current model we defined does not incorporate any notion of *word word* into the contextual representations of symbols or the encodings of the context $\mathbf{h}_t$. Currently, all operations composing the transformer are *position-agnostic*: The convex combination of the value vectors $\mathbf{V}$ will be the same, no matter the permutation of the vectors. The keys also cannot contain any positional information. To be able to take into account the word order in a transformer, we have to explicitly provide the positional information to the model. The simplest way to do this is to augment the static symbol encodings in the first transformer layer with *positional encodings*, in the form of vectors which can be added or concatenated to the static encodings of symbols.

Positional encoding

A **positional encoding** is a function $\mathbf{f}_{\text{pos}} : \mathbb{N} \to \mathbb{R}^D$

And very straightforward:

Position-augmented representation function

Let $\mathbf{e}' : \overline{\Sigma} \to \mathbb{R}^D$ be a symbol representation function and $\mathbf{f}_{\text{pos}} : \mathbb{N} \to \mathbb{R}^D$ a positional encoding. A **position-augmented** representation function of a symbol y_t in a string $\mathbf{y}$ is the representation function $\mathbf{e}'_{\text{pos}} : \overline{\Sigma} \to \mathbb{R}^D$ defined as

$$\mathbf{e}'_{\text{pos}}(y_t) \stackrel{\text{def}}{=} \mathbf{e}'(y_t) + \mathbf{f}_{\text{pos}}(t) \tag{113}$$

Multiple heads Importantly, the transformer introduced so far computes a single set of contextual representations - one for every input symbol (at every layer of the transformer). However, we can easily extend the model to compute *multiple* contextual representations for each symbol. This is done using so-called **multi-head attention**, where a single attention block is called an **attention head**. This increases the representation space of the individual symbols and thus enables the model to capture more information about the symbols and the sentence. The interpretation of computing multiple representations (one for each head) independently also invites the interpretations that each of the heads "focuses" on a separate aspect of the text. To be able to use the outputs of multi-head attention as inputs to the next block again, the outputs of the different attention heads are then concatenated and then projected down to the output size of a single attention block using an additional transformation.

Multi-Head Attention Block

Let $H \in \mathbb{N}$ be the number of attention heads, $Q_h(\cdot), K_h(\cdot)$, and $V_h(\cdot)$ be parameterized functions from $\mathbb{R}^{T\times D}$ to $\mathbb{R}^{T\times D}$ for $0 \leq h \leq H$, and $\mathbf{f}_H : \mathbb{R}^{T\cdot H\times D} \to \mathbb{R}^{T\times D}$ defined as

$$\texttt{MH-A}(\mathbf{X}) = \mathbf{f}_H(\text{concat}_{0\leq h<H}\Big(\text{softmax}(Q_h(\mathbf{X})K_h(\mathbf{X})^\top)V_h(\mathbf{X})\Big)) \tag{114}$$

Layer normalization As a final component of a transformer, we mention layer normalization, similar to the use of residual connections, represents a common trick in the deep learning space for ensuring more stable and reliable gradient-based learning, as such, it is not limited to transformers.

Layer Normalization

Let $\mathbf{x}, \boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^D$, and $\epsilon > 0$. The **layer normalization** function $\texttt{LN} : \mathbb{R}^D \to \mathbb{R}^D$ is defined as

$$\texttt{LN}(\mathbf{x}; \boldsymbol{\gamma}, \boldsymbol{\beta}) \stackrel{\text{def}}{=} \frac{\mathbf{x} - \bar{\mathbf{x}}}{\sqrt{\sigma^2(\mathbf{x}) + \epsilon}} \odot \boldsymbol{\gamma} + \boldsymbol{\beta} \tag{115}$$

where $\bar{\mathbf{x}}$ refers to the mean of the vector $\mathbf{x}$ (and is subtracted from all elements of $\mathbf{x}$ in the formulation above) and $\sigma^2(\mathbf{x})$ refers to the variance of elements $\mathbf{x}$. ϵ is added in the denominator to sure the stability if $\sigma^2(\mathbf{x}) << 1$

5 *Tokenization*

There aren't any good mathematical for a tokenizer, but we can state the follows:

Tokenizer

A tokenizer takes a string in Σ^* and maps it to words/tokens in Δ^*, where Δ is a finite set of words and UNK.

Good requirements:

- Tokenizer t has all of Σ^* in its pre-image.

An example of a tokenizer is whitespace tokenization. However there are some considerations

- if there are too many UNKs,
- punctuation split off,
- morphology.

The solution is data driven tokenizers. Why not set $\Sigma =$ all unicode characters? Bad empirically, harder to train. All modern tokenizers learn a segmentation

1. between word and character
2. not 100% linguistic

Byte-Pair Encoding Input: Σ, $C = \{x^{(m)}\}_{m=1}^{M} \subset \Sigma^*$. Return: $\Delta, t : \Sigma^* \to \Delta^*$ Steps:

1. Initialize Δ to Σ
2. Find the most frequent merge in C, where a merge m is a concatenation of two elements in Δ, so now $\Delta \leftarrow \Delta \cup m$

Example for **Merge**, consider the corpus:

$$\text{a a b a a \_ b b}$$

We being this with the character level, so at first we count as:

$$\text{| a | a | b | a | a | \_ | b | b |} \in \Delta^*$$

With this $\Delta^{(0)} = \{a, b, \_\}$. Now we can find the most frequent merge!

$$\text{| } \mathbf{a} \text{ | } \mathbf{a} \text{ | b | } \mathbf{a} \text{ | } \mathbf{a} \text{ | \_ | b | b |} \in \Delta^*$$

So we update Δ to become $\Delta^{(1)} = \Delta^{(0)} \cup \{\text{aa}\}$ as aa is the most common adjacent symbol. (ab occurs just once, ba just once, ...)

And hence, you get

$$\texttt{| a a | b | a a | \_ | b | b |} \in \Delta^*$$

Now they all occur an equal number of times, so we can choose arbitrarily, so we can get $\Delta^{(2)} = \Delta^{(1)} \cup \{\texttt{b b}\}$, and now just continue for a finite number of times.

Now, as we said t is a function that maps from $\Sigma^* \to \Delta^*$, what is the function? Where does t come from? Function t essentially replays the merges in the order they were added, fixing conflicts from left-to-right.

Spurious Ambiguity Suppose we had a string `aaba`, then we can segment it as `|a|a|b|a|` or `|aa|b|a|` which is fine since each segmentation all exist in Δ, but this introduces ambiguity that you do not want. Both segmentations are valid under BPE, meaning both segmentations exist in the set of learned tokens Δ However, the choice between these segmentations is not inherently clear, leading to ambiguity.

The pushforward $t : \Sigma^* \to \Delta^*$ looks like

$$p(x) = \sum_{\substack{y \in \Delta^* \\ y \in t^{-1}(x)}} p_{\Delta^k}(y) \tag{116}$$

where

$$t^{-1}(x) = \{y | y \in \Delta^*, t(x) = y\} \tag{117}$$

6 *Generation from Language Models*

A LM is a distribution p over Σ^*, but the end goal is to return text to the user. There stochastic and deterministic methods:

- Stochastic
  - Sampling $x \sim p$, efficient due to local normalization. So if we have
    $$p(x) = p(\text{EOS}|x) \prod_{t=1}^{T} p(x_t|x_{<t}) \tag{118}$$
    Then we can sample $x_1 \sim p(\cdot|\text{BOS})$, $x_2 \sim p(\cdot|x_1)$, $x_3 \sim p(\cdot|x_1x_2)$, $\dots$, the runtime for each of these is $\mathcal{O}(|\Sigma|)$. But why does this not solve the problem? This is because it **often generates strings users do not like**!
- Deterministic
  - Maximum $\text{argmax}_{x \in \Sigma^k} \log p(x)$, which is often intractable!

Greedy Search + Beam Search These are both local search techniques. In greedy search, $x_i = \text{argmax}_{x \in \Sigma^*} \log(x|x_1 \dots x_{i-1})$. Might not even halt, even for tight models, and can be made arbitrarily bad. On the other hand, beam search is a generalization of greedy search where we keep k paths, and at every step choose the best k continuations of these. Can also be made arbitrarily bad. Some considerations are when to stop, how to bucket (by timestep is natural but beware tokenization)

How to handle surface form competition? Many sentences have the same semantics, probability mass is spread more thinly among similar ones; does it still make sense to choose the highest probability string?

Top-k sampling is to only consider the top-k most probable next symbols in ancestral sampling. Nucleus Sampling only considers sampling the next symbols that are in $C(\boldsymbol{y}_{<t})$ which contains the top p% of the probability mass

Sampling adapter as function α that maps distributions over $\overline{\Sigma}$ to distributions over $\overline{\Sigma}$, as in $\tilde{p}(\cdot|\boldsymbol{y}_{<t}) = \alpha(p(\cdot|\boldsymbol{y}_{<t}))$. All thresholding algorithms are ancestral sampling on an adapted model.

Part II

Applications

7 *Transfer Learning*

Transfer learning is the idea of using knowledge gained from training on one task in order to solve on other tasks.

Transfer learning for language models

Consider a language model $p_{\text{LM}}(\boldsymbol{y}; \boldsymbol{\theta})$ over Σ^* trained on corpus $\mathcal{D} = \{\boldsymbol{y}^{(n)} \mid \boldsymbol{y}^{(n)} \in \Sigma\}_{n=1}^{N}$. Next, consider a target task $\mathcal{T}$, posed as learning a function $f : L \mapsto \mathcal{Y}$, for some input space $L \subseteq \Sigma^*$ and output space $\mathcal{Y}$. We say that transfer learning occurs if parameterizing f as $f_{\hat{\boldsymbol{\theta}}}$, with $\hat{\boldsymbol{\theta}} \subseteq \boldsymbol{\theta}$, allows for more efficient learning of $\mathcal{T}$ compared to initializing f as $f_{\boldsymbol{\theta}'}$, with $\boldsymbol{\theta}'$ being some set of parameters that are sampled randomly from some distribution.

Note that $\mathcal{T}$ can be any task, as long as it takes L as input. The network is trained on the source task and is called a **pretrained model**, and we refer to the learning process of that model as **pretraining**. The process of updating the weights of a pretrained model for a new task is called **fine-tuning**. Fine-tuning does not encapsulate all forms of transfer learning, as our definition does not necessitate udating the parameters of the language model. A related concept to transfer learning is **multi-task learning**, which is the idea of sharing learned information across multiple tasks. **In contrast to transfer learning**, the tasks are learned jointly rather than sequentially.

Language modeling is suitable for transfer learning since it is very easy to scale up language modeling data. We only require some corpus of text that approximately captures the domain we want to model. Since the input space of language modeling is the same as the output space it does not require any expensive labeling, it is a **self-supervised task**.

7.1 *ELMo*

ELMo was one of the first successful transfer learning models based on language modeling. It leverages the language modeling task to learn word representations, that is, vectors meant to represent the meaning of words. Whereas most previous approaches, such as word2vec and

GloVe, trained static word representations, the word representations in ELMo are context-dependent.

ELMo considers two separate language models, a (standard) forward language model $p_{\text{LM}}(y_t \mid \boldsymbol{y}_{<t})$ as well as a backward language model $p_{\text{LMB}}(y_t \mid \boldsymbol{y}_{>t}) \stackrel{\text{def}}{=} \prod_{t=1}^{T} p_{\text{LMB}}(y_t \mid y_{t+1}, \ldots, y_T)$. Both are implemented by stacking L layers of LSTMs, the parameters of which we refer to as $\overrightarrow{\boldsymbol{\theta}}$ and $\overleftarrow{\boldsymbol{\theta}}$ respectively. For a given input token y_t, the forward LSTM layers output context-dependent representations $\overrightarrow{\mathbf{h}}_{tl}^{\text{LM}}$, with $l \in [0, L]$ (and analogously, $\overleftarrow{\mathbf{h}}_{tl}^{\text{LM}}$ for the LSTM layers of the backward language model). The deepest representations $\overrightarrow{\mathbf{h}}_{tl}^{\text{LM}}$ and $\overleftarrow{\mathbf{h}}_{tl}^{\text{LM}}$ are fed to a softmax layer to predict the forward and backward probabilities of y_t. In addition to $\overrightarrow{\boldsymbol{\theta}}$ and $\overleftarrow{\boldsymbol{\theta}}$, the parameters for token representations and the softmax layer (denoted together as $\boldsymbol{\theta}'$) are tied between the two networks. All parameters are optimized jointly by maximizing the log likelihoods of the forward and backward models:

$$\mathcal{L}_{\texttt{ELMo}}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{t=1}^{T} \log p_{\text{LM}}(y_t{}^{(n)} \mid \boldsymbol{y}_{<t}^{(n)}; \overrightarrow{\boldsymbol{\theta}}, \boldsymbol{\theta}') + \log p_{\text{LMB}}(y_t{}^{(n)} \mid \boldsymbol{y}_{>t}^{(n)}; \overleftarrow{\boldsymbol{\theta}}, \boldsymbol{\theta}') \tag{119}$$

Now in order to fine-tune for a specific task, we can use the context-specific representations $\mathbf{h}_{tl}^{\text{LM}} = [\overrightarrow{\mathbf{h}}_{tl}^{\text{LM}}; \overleftarrow{\mathbf{h}}_{tl}^{\text{LM}}]$. One could simply take the last layer representations $\mathbf{h}_{tl}^{\text{LM}}$ and use those as input to a separate model that is fine-tuned on another task. The original paper additionally experimented with learning a task-specific representation as a scaled convex combination over the hidden representations, as follows:

$$\mathbf{ELMo}_t^{\text{task}} = \gamma^{\text{task}} \sum_{l=0}^{L} s_l^{\text{task}} \mathbf{h}_{tl}^{\text{LM}}, \tag{120}$$

where $\sum_{l=0}^{L} s_l^{\text{task}} = 1$ are the outputs of a softmax function over the hidden representations, and $\gamma^{\text{task}} \in \mathbb{R}$ is meant to scale the representations.

7.2 *BERT*

After the success of ELMO, pre-trained BERT (Bidirectional Encoder Representations from Transformers), a Transformer-based bidirectional masked language model. BERT is pre-trained with the masked language modeling objective on large scale text corpus. After pretraining, BERT can be fine-tuned to perform different NLP task without the need of design task-specific architectures. BERT advanced the state-of-the-art of many NLP benchmarks at the time it was proposed.

In this section, we first introduce the pretraining objective of BERT. We then describe BERT model architecture and the pretraining and fine-tuning paradigm.

BERT Pre-training Objectives BERT is pretrained jointly with two self-supervised tasks: **masked language modeling** and **next sentence prediction**. This is done at the same time. The losses for each task are computed and summed together:

$$\mathcal{L} = \mathcal{L}_{\text{MLM}} + \mathcal{L}_{\text{NSP}} \tag{121}$$

Masked Language Modeling. Masked language modeling (MLM) adapted this idea as a novel pre-training task to overcome the drawback of the standard unidirectional LM. In the masked language modeling setup, the goal is to predict the omitted token from a piece of text that constitutes a logical and coherent completion. For example, in the piece of text "The students `[MASK]` to learn about language models", we predict *want* or *like* with high probability for the `[MASK]` position. The goal of masked language modeling is to approximate the probability distribution over tokens in our vocabulary as the original token at a given masked position. Similarly to the standard language modeling objective, we can choose model parameters by optimizing for the log-likelihood of a dataset $\mathcal{D}$. Albeit in this case, the words at a percentage of randomly-chosen positions in $\mathcal{D}$ are replaced with `[MASK]` and the model is given **both** sides of context around the masked token in order to make its prediction:

$$\mathcal{L}_{\texttt{MLM}}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{t=1}^{T} \log({y_t}^{(n)} \mid \boldsymbol{y}_{<t}^{(n)}, \boldsymbol{y}_{>t}^{(n)};) \mathbb{1}\{{y_t}^{(n)} = \texttt{[MASK]}\} \tag{122}$$

However, this pretraining method will create a mismatch between the pretraining phase and the fine-tuning phase because the mask token does not appear during the fine-tuning phase. Empirically, to deal with this issue, it was proposed to use a special `[MASK]` token 80% of the time, a random token 10% of the time and the original token 10% of the time to perform masking.

Next Sentence Prediction. The next sentence prediction objective is included to enable the model to capture the relationships between two consecutive sentences. This is important because many NLP tasks require understanding the relationship between different text inputs (e.g., question and context for the question answering task). By pretraining the model to predict whether a given sentence follows another

sentence, we can help the model learn the dependencies and relationships between sentences.

The next sentence prediction objective is implemented as follows. Given a pair of sentences, denoted as A and B, the model is trained to predict whether B is the next sentence that follows A. This is done by feeding the two sentences as input to the model, along with a special [CLS] token that serves as the input representation of the entire sequence. The model then generates a binary output that indicates whether B is the next sentence.

BERTs Architecture BERT's model architecture is a multi-layer bidirectional Transformer encoder. BERT is a Transformer encoder model. It takes a sequence of text tokens as input and produces their contextualized representations and (optionally) predictions. We denote the number of layers (i.e., Transformer blocks) as L, the hidden size as H, and the number of self-attention heads as A. In all cases the feed-forward/filter size is set to be $4H$, i.e., 3072 for the $H = 768$ and 4096 for the $H = 1024$.

To make BERT handle a variety of down-stream tasks, BERT input representation is able to unambiguously represent both a single sentence and a pair of sentences (e.g., $\langle$ Question, Answer $\rangle$) in one token sequence. BERT use WordPiece embeddings with a 30,000 token vocabulary. The first token of every sequence is always a special classification token (`[CLS]`). The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks. Sentence pairs are packed together into a single sequence. We differentiate the sentences in two ways. First, we separate them with a special token (`[SEP]`). Second, we add a learned embedding to every token indicating whether it belongs to sentence `A` or sentence `B`.

7.3 BERT Variants

RoBERTa RoBERTa (Robustly Optimized BERT Approach) is an optimized version of BERT. RoBERTa uses the same model architecture of BERT but with some improvements to the pre-training process. RoBERTa uses a larger pre-training corpus, larger batch size, longer training time, longer input sequence lengths, larger batch size, and a dynamic masking strategy during pre-training.

Specifically, RoBERTa adds CC-NEWs and OpenWebText to the original BERT pre-training data. The resulting corpus contains over 160 GB of text and approximately 2.5 billion word pieces. RoBERTa is pre-trained with a batch size of 8,000 sequences of 512 tokens for 500k steps, resulting in much more computation compared to BERT. The

dynamic masking strategy, which randomizes the masking pattern in each epoch and helps the model to learn more robust representations of the input text. These modifications allows it to surpass BERT and achieve state-of-the-art performance on several NLP tasks at the time.

8 Parameter Efficient Finetuning

Pretrained language models are used in a wide range of NLP tasks. When the model size becomes larger and larger, it is more and more difficult to tune the model with limited size of annotated data. Overfitting easily happens, and the cost of model tuning is quite expensive. Thus, to avoid above issues, various parameter-efficient tuning methods are proposed. In the following sections, we will first introduce partially fine-tuning techniques. They are simple but effective, which is widely used in Transformer model tuning. Besides selecting part of parameters for tuning, another line of work explores how to keep the model frozen, and add extra tuned parameters. These newly added and tuned modules are called adapters.

8.1 Partial Fine-tuning

This type of tuning methods fine tune few inherent parameters while leaving the majority of parameters unchanged in model adaptation. This approach does not seek to change the internal structure of a model, but to optimize a small number of internal parameters to solve particular tasks. Generally, such specifications could be implemented based on heuristics or training supervision.

Heuristic Specification Specification-based methods do not introduce any new parameters in the model, but directly specify part of the parameters to be optimized. The idea is simple but very effective, **only finite-tuning 1/4 of the final layers** of BERT and RoBERTa could product 90% of the performance of full parameter fine-tuning.

BitFit only optimizes the **bias** term in the model while freezing all the other parameters, and the model could stil reproduce over 95% performance on several benchmarks. The biased term exists in both attention mechanism and in the MLP feedforward layer. BitFit can finetune only **two bias components** (the query and the middle of MLP bias terms), amounting to half of the bias parameters in the model, and only 0.04% of all the model parameters. The corresponding bias terms are coloured in red, other bias terms are in blue.

$$\texttt{Query in Attention}: \quad \mathbf{Q}(\mathbf{x}) = \mathbf{W}_q\mathbf{x} + \mathbf{b}_q \tag{123}$$

$$\mathbf{K}(\mathbf{x}) = \mathbf{W}_k\mathbf{x} + \mathbf{b}_k \tag{124}$$

$$\mathbf{V}(\mathbf{x}) = \mathbf{W}_v\mathbf{x} + \mathbf{b}_v \tag{125}$$

$$\texttt{MLP in Feedfoward}: \quad \mathbf{h}_2 = \text{Dropout}(\mathbf{W}_1 \cdot \mathbf{h}_1 + \mathbf{b}_1) \tag{126}$$

$$\mathbf{h}_3 = \mathbf{g}_{LN_1} \odot \frac{(\mathbf{h}_2 + \mathbf{x}) - \mu}{\sigma} + \mathbf{b}_{LN_1} \tag{127}$$

$$\mathbf{h}_4 = \text{GELU}(\mathbf{W}_2 \cdot \mathbf{h}_3 + \mathbf{b}_2) \tag{128}$$

$$\mathbf{h}_5 = \text{Dropout}(\mathbf{W}_3 \cdot \mathbf{h}_4 + \mathbf{b}_3) \tag{129}$$

$$\text{out} = \mathbf{g}_{LN_2} \odot \frac{(\mathbf{h}_5 + \mathbf{h}_3) - \mu}{\sigma} + \mathbf{b}_{LN_2} \tag{130}$$

Empirical results in BitFit also show that even if we use a small random set of parameters for tuning (which will obviously degrade the performance), the model could still yield passable results on the GLUE benchmark.

Unfortunately, the work only applies this trick to small-scale models, and there is no guarantee that randmly choosing some parameters to be tuned would remain competitive for larger models. Another valuable observation is that different bias terms may have different functionalities during model adaption.

Learn the Specification Rather than manually or heuristically specify which parameters to be updated, an alternative is to learn such specifications. *Diff pruning* parameterized the finetuned model parameters as the summation of the pretrained parameters and the difference vector as

$$\theta_{FT} = \theta_{LM} = \delta_{Diff} \tag{131}$$

Hence, the key issue is to encourage the difference vector δ_{Diff} to be as sparse as possible. This work regularizes the vector by a differentiable approximation to the L_0-norm penalty as $||\delta_{Diff}||_0$ to achieve the goal of sparsity. Practically, because new parameters to be optimized are introduced in the learning phase, *Diff pruning* takes up more GPU memory than full parameter fine-tuning, which may establish barriers in the application on large language models.

8.2 Adapter Tuning

The third idea we are going to explore is adapter tuning, which inserts small modules called **adapters** to a model. Adapters can be any network architectures and can be placed anywhere in the model, but often is to place a two-layer FFN with a bottleneck after each sublayer

(including both the MHSA sublayer and the FFN sublayer) within the transformer.

$$\mathbf{h} \leftarrow \mathbf{h} + f(\mathbf{h}\mathbf{W}_{\text{down}})\mathbf{W}_{\text{up}} \tag{132}$$

where f is a nonlinear activation function. Some observations:

- **Comparable results** to full finetuning. Adapters achieve a mean GLUE score of 80.0, compared to 80.4 achieved by full fine-tuning.
- The optimal adapter size varies per dataset - 256 is chosen for MNLI, 8 is chosen for RTE (smaller dataset).
- Restricting adapter size to 64, leads to a small decrease in accuracy to 79.6.

Prefix Tuning Prefix tuning is another PEFT method, but different from others, it has its roots in prompting, which will be discussed in detail later. Freeze Transformer parameters and optimize (activations corresponding to) a prefix for each task (red prefix blocks). We only need to store the prefix representations for each task, making prefix-tuning modular and space-efficient.

8.3 LoRA (Low Rank Adaptation of Models)

9 *Prompting and Zero-shot Inference*

In a traditional supervised learning setting, an output $\boldsymbol{y}$ is generated given the input $\boldsymbol{x}$ and the model parameters θ using the modeling objective $P(y|x;\theta)$. However, for many tasks, the supervised data is unavailable making the training process difficult/impossible. Prompting enables models to circumvent the training issue by learning an LM that models the probability $P(\boldsymbol{x};\theta)$ of text $\boldsymbol{x}$ itself, and using this probability to predict $\boldsymbol{y}$, reducing or obviating the need for large supervised datasets. In other words, prompting is non-invasive: it does not introduce large amounts of additional parameters or require direct inspection of a model's representations. It can be thought of as a lower bound on what the model "knows" and can be used to extract information from LM.

To prompt an LM, it is important to map the input $\boldsymbol{x}$ to a prompt $\boldsymbol{x}'$. A prompting function $f_{\text{prompt}}(\cdot)$ is applied to modify the input text $\boldsymbol{x}$ into a prompt $\boldsymbol{x}' = f_{\text{prompt}}(\boldsymbol{x})$. Next, a template is defined that consists of two slots: an input slot `[X]` for input $\boldsymbol{x}$ and an answer slot `[Z]` for any generated answer $\boldsymbol{z}$ that may or may not be mapped into $\boldsymbol{y}$ depending on the use case. Next, the highest-scoring $\hat{\boldsymbol{z}}$ that maximizes the score of the LM is taken ($\boldsymbol{z}$ can take all possible values from the vocabulary for generation tasks but can also be modified for controlled classification/generation tasks).

A function $f_{\text{fill}}(\boldsymbol{x}', \boldsymbol{z})$ fills in the location `[Z]` in prompt $\boldsymbol{x}'$ with the potential answer $\boldsymbol{z}$ by searching over the set of all potential answers by calculating the probability of their corresponding filled prompts using a pre-trained LM $P(\cdot;\theta)$.

$$\hat{\boldsymbol{z}} = \underset{\boldsymbol{z}\in\mathcal{Z}}{\text{search}}\, P(f_{\text{fill}}(\boldsymbol{x}', \boldsymbol{z});\theta). \tag{133}$$

This search function could be an *argmax* search that searches for the highest-scoring output or various *sampling* techniques that randomly generate outputs following the probability distribution of the LM.

9.1 *Prompt Engineering*

Given a task, multiple prompts can work. However, finding the most effective prompts is desired in order to unlock the full potential of the LM. *Prompt Engineering* is the process of designing a prompting function $f_{\text{prompt}}(\boldsymbol{x})$ that results in the most effective performance for the given task.

Manual Prompts The straightforward yet somewhat effective technique is to design the prompts for a given task manually. Since the number

of required prompts is usually very less, designing manual prompts gives more control and flexibility to the user.

However, there are several problems with this approach: creating and experimenting with these prompts is an art that takes time and experience, especially for more complicated tasks like multi-step reasoning, and even experienced prompt designers may fail and find optimal prompts manually.

Automated Prompts Searching for automated prompts can be further divided into *discrete prompts* or *continuous prompts*.

Discrete prompts (aka *hard prompts*), as the name suggests, automatically searches for prompts in a discrete space (the prompts are usually text strings corresponding to natural language). Various methods have been proposed in this line of work and we explore a few approaches here.

Some initial work on automating discrete prompt design with moderate success:

- **Mine prompt candidates from a large corpus**, search for strings in a large text corpus that contains both training inputs x and outputs y. Then, you identify the middle words or dependcy paths between them.
- Use the middle words/paths as templates in the form of `[X] middle words [Z]`.
- **Paraphrase approach** translating the prompt into another language and back, use a thesarus to replace words, train a neural prompt rewriter designed/trained with the objective of improving the accuracy of systems that use the prompt.
- Training a text generation model for generating prompts

Continuous prompts Prompt construction aims to enable an LM to perform a specific task effectively, and it is not imperative for the prompt to be limited to natural language that can be interpreted by humans, therefore, some approaches focus on *continuous prompts* (aka. soft prompts) that prompt the model directly in its embedding space.

One such approach is *Prefix Tuning* that prepends a sequence of continuous task-specific vectors to the input while keeping the LM parameters frozen. Mathematically, this consists of optimizing over the following log-likelihood objective given a trainable prefix matrix M_ϕ and a fixed pre-trained LM parameterized by θ.

$$\max_{\phi} \log P(\boldsymbol{y}|\boldsymbol{x};\theta;\phi) = \max_{\phi} \sum_{y_i} \log P(y_i|h_{<i};\theta;\phi) \tag{134}$$

In Eq. 134, $h_{<i} = [h^{(1)}_{<i}; \cdots ; h^{(n)}_{<i}]$ is the concatenation of all neural network layers at time step i. It is copied from M_ϕ directly if the corresponding time step is within the prefix (h_i is $M_\phi[i]$), otherwise it is computed using the pre-trained LM.

9.2 *Advanced Prompting*

Zero- and Few-Short Inference Prompting methods can often be used without *any* explicit training of the language model (LM) for the downstream task. One can take an LM that has been trained to predict the probability of text $P(\boldsymbol{x})$ and apply it as-is to fill the cloze or prefix prompts that define the task. This is traditionally referred to as the *zero-shot* setting, as there is no training data available for the task of interest.

However, in the scenario where a limited number of labeled examples are available or can be easily annotated, it is possible to augment the prompt to include this additional information. This setting is referred to as *few-shot* inference and consists of including a small collection of input-out exemplars. These exemplars are input-output pairs that serve as demonstrations of the behavior that one would like the LM to emulate. By using these pairs as a guide, the large LMs can learn to carry out the desired task. This simple idea was shown to for example, we can augment the standard prompt "France's capital is `[X]` ." by prepending a few examples such as "Great Britain's capital is London . Germany's capital is Berlin . France's capital is `[X]`". Note that few-shot inference does not involve any parameter updates.

Chain of Thought Prompting Multiple studies investigating the reasoning capabilities of large LMs, highlighted how eliciting the model to produce a step-by-step solution of a problem can lead to a more accurate final answer. Combining this idea with the intuition of including a set of demonstrations in the prompt, introduced the concept of chain of thought (CoT) prompting. Given a question, a *chain of thought* is a coherent sequence of reasoning steps that leads to a final answer. One of the intuitions is that by allowing the model to generate a step-by-step solution, we let the model apply computation proportional to the problem difficulty level (more tokens are generated for problems with more steps).

Problem Decomposition Chain-of-thought prompting has shown impressive results on a variety of natural language reasoning tasks. However, its effectiveness decreases when a given task is more challenging than the examples provided in the prompts. To address this issue, problem decomposition has been proposed both for training and as

a prompting strategy. In this method, a complex problem is broken down into smaller, more manageable sub-problems that are tackled one at a time. The solution to each subproblem helps to solve the next, allowing for a sequential problem-solving process. An example of this prompting technique, often referred to as Least to Most prompting,

Self-consistency Greedy decoding (setting with temperature $T = 0$) can be considered the most naive decoding strategy, where an LM is given a prompt with or without in-context examples and an output is generated. However, this limits the creativity of the LM by generating only one greedy sample per input. An extension to the decoding strategy called *self-consistency*, which provides an alternative to simple greedy decoding by generating a variety of outputs (with $T > 0$) instead of just following the most likely one.

Taking an example of the task of solving reasoning problems, the most consistent answer is defined by sampling n candidate pairs of reasoning paths and their corresponding final numerical answer (r_i, a_i) and marginalizing over r_i by taking a majority vote over a_i:

$$\arg\max_a \sum_{i=1}^{n} \mathbb{1}(a_i = a). \tag{135}$$

The same idea could be used for other tasks by taking the most consistent answer over the range of sampled outputs. The *self-consistency* approach argues that a task, say a reasoning task, often allows for several different lines of thought that converge on the same correct solution. In this case, the most frequently generated solution is more likely to be the correct one.

10 Vision Language Models

Vision-and-language tasks, by definition, should include both vision and language modalities in their inputs and outputs. VL tasks can be grouped into three categories: **Image-Text Tasks, CV Tasks as VL Tasks**, and **Video-Text Tasks**.

10.1 Components of a Vision Language Model

Text Encoder VLMs first segment the input sentence into a sequence of subwords and then insert two special tokens at the beginning and the end of the sentence to generate the input text sequence. After we obtain the text embeddings, existing works either feed them directly to the multimodal fusion module, or to several text-specific layers before the fusion. For the former, the fusion module is typically initialized with BERT, and the role of text encoding and multimodal fusion is therefore entangled and absorbed in a single BERT model, and in this case, we consider text encoder as the word embedding layer.

In a nutshell, no matter what text encoder is used, the input text is represented as a set of feature vectors $w = \{w_1, \cdots, w_N\}$

Vision Encoder Then, we have a **vision encoder**. There are three types there are three types of vision encodes, which we will go into more detail.

- **Object Detectors (OD)**. The most widely used object detector for VL research is the Faster R-CNN pre-trained on the Visual Genome (VG) datasat as in BUTD. Start with a pretrained Object detector e.g. R-CNN or Faster R-CNN. Use visual features as well as location features $[x_1, y_1, x_2, y_2, w, h, w \times h]$ (normalized coordinates, width, height and area). Both visual and location features are combined, e.g., fed through a fully-connected layer.
- **Convolutional Neural Networks**. Shift invariant architecture for image representation. Convolution + RELU, Convolution layer puts the input image through a set of convolutional filters, each of which activates certain features in the image. RELU, a non-linear activation function - allows for effective training by mapping negative values to zero and maintaining positive values. Pooling - simplifies the output by performing a down-sampling.
- **Vision Transformers**. Create image tokens: Split image into image patches, map them into vectors and linearly project them to patch embeddings. [CLS]: Add a learnable special token [CLS] embedding to the sequence.

  In a nutshell, no matter what vision encoder is used, the input im-

age is represented as a set of feature vectors $\mathbf{v} = \{\mathbf{v}_1, \cdots, \mathbf{v}_M\}$. VLMs can be categorized into non end-to-end models, which use an OD model to get vision features for the model, and end-to-end models that directly take raw images as input.

Multimodal Fusion For *dual encoders* like CLIP and ALIGN, fusion is essentially computing the similarity between representation in the two modalities, which is typically performed via a dot-product between two global image and text feature vectors. For *fusion encoder*, it takes both $v = \{v_1, \cdots, v_M\}$ and $w = \{w_1, \cdots, w_N\}$ as input, and learns contextualized multimodal representations denoted as $\tilde{v} = \{\tilde{v}_1, \cdots, \tilde{v}_M\}$ and $\tilde{w} = \{\tilde{w}_1, \cdots, \tilde{w}_N\}$. There are mainly two types of fusion modules, namely, *merged attention* and *co-attention*.

- In a **merged attention** module, the text and visual features are simply concatenated together, and then fed into a single Transformer block. This design has been used in many previous works, such as VisualBERT, Unicoder-VL, VL-BERT, UNITER, OSCAR, VinVL, ViLT, GIT.
- In a **co-attention** module, on the other hand, the text and visual features are fed into different Transformer blocks independently, and techniques such as cross-attention are used to enable cross-modal interaction. Most ViT-based models adopts co-attention module since the image sequence can be very long and doing merged attention can be very computationally inefficient.

10.2 Vision-Language Models: Pre-training Objectives

Masked Language Modeling (MLM). The MLM objective is first introduced in language model pre-training. In VLP, MLM with image-text pairs has also proven to be useful. In MLM, given an image-text pair, we randomly mask out the input words with probability of 15%, and replace the masked ones $\tilde{\mathbf{w}}_\mathbf{m}$ with special token `[MASK]`.[7] The goal is to predict these masked tokens based on their surrounding words $\tilde{\mathbf{w}}_{\setminus \mathbf{m}}$ and the paired image $\tilde{\mathbf{v}}$, by minimizing the negative log-likelihood:

$$\mathcal{L}_{\text{MLM}}(\theta) = -\mathbb{E}_{(\tilde{\mathbf{w}}, \tilde{\mathbf{v}}) \sim D} \log P_\theta(\tilde{\mathbf{w}}_\mathbf{m} | \tilde{\mathbf{w}}_{\setminus \mathbf{m}}, \tilde{\mathbf{v}}), \tag{136}$$

where θ denotes the trainable parameters. Each pair $(\tilde{\mathbf{w}}, \tilde{\mathbf{v}})$ is sampled from the whole training set D. There are several MLM variants used in VLP.

[7] Following BERT, this 15% is typically decomposed into 10% random words, 10% unchanged, and 80% `[MASK]`.

Image-Text Matching (ITM). In ITM, given a batch of matched or mismatched image-caption pairs, the model needs to identify which images and captions correspond to each other. Most VLP models treat image-text matching as a binary classification problem. Specifically, a

special token (*i.e.*, `[CLS]`) is appended at the beginning of the input sentence to learn a global cross-modal representation. We then feed the model with either a matched or mismatched image-caption pair $\langle \tilde{\mathbf{v}}, \tilde{\mathbf{w}} \rangle$ with equal probability, and a classifier is added on top of the `[CLS]` token to predict a binary label y, indicating whether the sampled image-caption pair is matched. Specifically, denote the output score as $s_\theta(\tilde{\mathbf{w}}, \tilde{\mathbf{v}})$, We apply the binary cross-entropy loss for optimization:

$$\mathcal{L}_{\text{ITM}}(\theta) = -\mathbb{E}_{(\tilde{\mathbf{w}},\tilde{\mathbf{v}})\sim D}[y \log s_\theta(\tilde{\mathbf{w}}, \tilde{\mathbf{v}}) + (1-y)\log(1 - s_\theta(\tilde{\mathbf{w}}, \tilde{\mathbf{v}}))]) \,. \tag{137}$$

Besides randomly sampling a negative image-text pair, harder negative pairs can also be mined from an image-text contrastive loss introduced below, which has been shown to be effective in improving the downstream performance.

Image-Text Contrastive Learning (ITC). Specifically, given a batch of N image-text pairs, ITC aims to predict the N matched pairs from all the N^2 possible image-text pairs. With a little bit abuse of notation, let $\{v_i\}_{i=1}^N$ and $\{w_i\}_{i=1}^N$ denote respectively the normalized image vectors and text vectors in a training batch. To compute image-to-text and text-to-image similarities, we have:

$$s_{i,j}^{i2t} = v_i^\top w_j, \; s_{i,j}^{t2i} = w_i^\top v_j \,, \tag{138}$$

$$\mathcal{L}_{\text{ITC}}^{i2t}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp(s_{i,i}^{i2t}/\sigma)}{\sum_{j=1}^{N}\exp(s_{i,j}^{i2t}/\sigma)} \,, \tag{139}$$

$$\mathcal{L}_{\text{ITC}}^{t2i}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp(s_{i,i}^{t2i}/\sigma)}{\sum_{j=1}^{N}\exp(s_{i,j}^{t2i}/\sigma)} \,, \tag{140}$$

Masked Image Modeling (MIM). Similar to the MLM objective, researchers have studied various masked image modeling (MIM) tasks for pre-training. Specifically, the model is trained to reconstruct the masked patches or regions $\tilde{\mathbf{v}}_{\mathbf{m}}$ given the remaining visible patches or regions $\tilde{\mathbf{v}}_{\backslash \mathbf{m}}$ and all the words $\tilde{\mathbf{w}}$ as

$$\mathcal{L}_{\text{MIM}}(\theta) = \mathbb{E}_{(\tilde{\mathbf{w}},\tilde{\mathbf{v}})\sim D} P_\theta(\tilde{\mathbf{v}}_{\mathbf{m}} | \tilde{\mathbf{v}}_{\backslash \mathbf{m}}, \tilde{\mathbf{w}}) \,. \tag{141}$$

Part III

Security